

Knihovna pro určení vzájemně podobných fotografií vhodného pro produkční provoz

Bc. Dobroslav Pelc

Diplomová práce
2018

 Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

*** Nascanované zadání, strana 1 ***

*** Nascanované zadání, strana 2 ***

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této současti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....

podpis autora

ABSTRAKT

Cílem této práce je analýza možností pro určení vzájemně podobných fotografií. Na základě analýzy, srovnávacích a zátěžových testů je vybrán nejvhodnější návrh řešení pro potřeby reálného produkčního provozu. Výsledná komponenta je realizována formou distribuované služby.

Klíčová slova: podobnost, zastupitelnost, distribuce, křízová korela, konvoluce, Fourierova transformace, světlostní matice

ABSTRACT

The aim of this work is to analyze the possibilities for identifying similar photos. Based on analysis, benchmarking and stress testing, the most suitable solution design is chosen for real production use. The resulting component is implemented in the form of a distributed service.

Keywords: similarity, replaceability, distribution, cross corelation, convolution, Fourier transform, brightness matrix

Zde je místo pro případné poděkování, motto, úryvky knih, básní atp.

OBSAH

ÚVOD	10
I TEORETICKÁ ČÁST	10
1 KLASIFIKACE ŘEŠENÝCH OBLASTÍ	12
1.1 KONSTRUKTIVNÍ ZMĚNY FOTOGRAFIE.....	12
1.1.1 Návrh řešení	12
1.1.2 Oblast zájmu	12
1.2 DESTRUKTIVNÍ ZMĚNY FOTOGRAFIE	12
1.2.1 Návrh řešení	13
1.2.2 Oblast zájmu	13
1.3 KOMBINACE KONSTRUKTIVNÍCH A DESTRUKTIVNÍCH ZMĚN	13
1.3.1 Návrh řešení	13
1.3.2 Oblast zájmu	13
1.4 VÝBĚR REPREZENTATIVNÍHO VZORKU.....	13
1.4.1 Návrh řešení	13
1.4.2 Oblast zájmu	14
2 KOEFICIENT PODOBNOSTI DVOU FOTOGRAFIÍ	15
2.1 ZMĚNA VELIKOSTI FOTOGRAFIE	15
2.2 PŘEVOD NA SVĚTLOSTNÍ MATICI	16
2.3 PŘEVOD DO-Z VLNOVÉHO SPEKTRA	17
2.4 KŘÍŽOVÁ KORELACE	17
2.5 VÝPOČET VÝSLEDNÉHO KOEFICIENTU	19
3 KOEFICIENT ZASTUPITELNOSTI DVOU FOTOGRAFIÍ	20
3.1 DISKRÉTNÍ 2D KONVOLUCE	20
3.2 KONVOLUCE A VOLBA JÁDRA	20
II ANALYTICKÁ ČÁST	21
4 BRAINSTORMING	23
4.1 AKADEMICKÝ KRUH	23
4.2 PROFESNÍ KRUH	23
5 DIFERENČNÍ ANALÝZA (GAP ANALÝZA)	24
5.1 POPIS SOUČASNÉHO STAVU.....	24
5.2 POPIS CÍLOVÉHO STAVU	24
5.2.1 Nefunkční požadavky	24
5.3 ROZDÍLY	24

5.4	NÁVRH VARIANT K DOSAŽENÍ CÍLE	25
5.4.1	Výpočet KoP a KoZ na CPU produkčního serveru	25
5.4.2	Výpočet KoP a KoZ na CPU produkčního serveru s paralelizací na GPU	25
5.4.3	Výpočet koeficientů na PC farmě	25
5.5	ZHODNOCENÍ VARIANT	25
5.5.1	Výpočet KoP a KoZ na CPU produkčního serveru	26
5.5.2	Výpočet KoP a KoZ na CPU produkčního serveru s paralelizací na GPU	26
5.5.3	Výpočet koeficientů na PC farmě	27
6	BENCHMARKING.....	28
6.1	TESTOVACÍ PROSTŘEDÍ	28
6.1.1	Použitý HW	28
6.1.2	Použitý SW	28
6.2	VÝSLEDKY TESTOVÁNÍ.....	31
6.2.1	Výpočet koeficientů na CPU	32
6.2.2	Výpočet koeficientů na CPU s paralelizací na GPU	32
6.2.3	Výpočet koeficientů na PC farmě	33
III	PROJEKTOVÁ ČÁST	34
7	PŘÍPRAVA PROJEKTU.....	36
7.1	HW ARCHITEKTURA	36
7.1.1	Pohled na aktuální HW mapu	36
7.1.2	Pohled na plánovanou HW mapu	36
7.2	PОŽАДАВКЫ	37
7.2.1	Funkční požadavky	37
7.2.2	Nefunkční požadavky	37
7.3	PŘÍPADY POUŽITÍ	37
7.3.1	Aktéři	37
7.3.2	Případy užití.....	38
7.4	MODELY	38
7.4.1	Model tříd	38
7.4.2	Model služeb.....	39
7.5	SEKVENČNÍ DIAGRAM DISTRIBUOVANÉ SLUŽBY	39
8	SERVEROVÁ STRANA DISTRIBUOVANÉ SLUŽBY	41
8.1	DATABELIZACE FOTOGRAFIÍ	41
8.1.1	Výpočet poměrného počtu hran	41

8.2	FRONTA NEZPRACOVANÝCH OBRÁZKŮ	45
8.3	SERVLET PRO STAŽENÍ OBRÁZKŮ	45
8.4	SOAP API	46
8.4.1	Autentizace.....	46
8.4.2	Distribuce požadavků na výpočet KoP	48
9	KLIENTSKÁ (VÝKONNÁ) STRANA DISTRIBUOVANÉ SLUŽBY ...	49
	ZÁVĚR	52
	SEZNAM POUŽITÉ LITERATURY	53
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	56
	SEZNAM OBRÁZKŮ	57
	SEZNAM TABULEK	58
	SEZNAM ZDROJOVÉHO KÓDU	59
	SEZNAM PŘÍLOH	60

ÚVOD

První odstavec pod nadpisem se neodsazuje, ostatní ano (pouze první řádek, odsazení vertikální mezy odstavci je typycké pro anglickou sazbu; czech babel toto respektuje, netřeba do textu přidávat jakékoliv explicitní formátování, viz ukázka sazby tohoto textu s následujícím odstavcem).

Formátování druhého odstavce. Text text.

I. TEORETICKÁ ČÁST

1 Klasifikace řešených oblastí

Pro porovnání dvou fotografií a jejich vzájemné vyhodnocení jako podobné či nikoliv bude zaveden koeficient podobnosti fotografií (dále jen KoP). Postupně budou popsány všechny silné i slabé stránky pro možnosti výpočtu KoP. Základem klasifikace je vymezení rozdílů mezi vzorovou a referenční fotografií, které definují konstruktivní a destruktivní změny.

1.1 Konstruktivní změny fotografie

Konstruktivní změny fotografie jsou takové změny, které jsou viditelné pro lidské oko, avšak nemění zásadně charakter obrázku pro strojové zpracování. Možné změny (případně jejich kombinace) jsou rozvedeny níže.

1.1.1 Návrh řešení

Pro porovnání dvou podobných fotografií, které obsahují pouze konstruktivní změny, plně postačí křížová korelace, která bude urychlena (v rámci optimalizace HW času) pomocí diskrétní Fourierovy transformace [10].

1.1.2 Oblast zájmu

Tuto numerickou metodou lze detektovat většinu nejčastějších záměrných modifikací fotografií (např. automatické doostření a následné uložení jako kopie). Lze sem zahrnout jak změny vlastností vázané na původní fotografiu, tak drobné změny v původním obsahu fotografie.

Změny vlastností

- Změna sytosti barev
- Změna kontrastu
- Změna jasu

Změny obsahu

- Vodotisk
- Logo
- Šum

1.2 Destruktivní změny fotografie

Destruktivní změny fotografie jsou viditelné pro lidské oko, ale pouze vjemem lidského oka je často problém tyto fotografie bezpečně prohlásit za podobné. Ještě horší situace je

u strojového zpracování, pro které se výrazně mění charakter porovnávaných fotografií.

1.2.1 Návrh řešení

Výpočet Hausdorfovy vzdálenosti mezi konvexními polyedry, které reprezentující hrany v obrazu [15].

1.2.2 Oblast zájmu

- Změna komprese (rozmazená fotografie)
- Změny rozlišení
 - Ořez (v jedné nebo obou dimenzích)
 - Deformace (v jedné nebo obou dimenzích)

1.3 Kombinace konstruktivních a destruktivních změn

Kombinace konstruktivních a destruktivních změn je vždy potřeba vyhodnotit nad konkrétním případem. Platí také, že jsou velmi obtížně řešitelné. V závislosti na míře změn lze často rozpoznat KoP stejně jako u čistě konstruktivních změn.

1.3.1 Návrh řešení

Redukce fotografie na její prahovou velikost jako příprava na křížovou korelaci viz konstruktivní změny [14].

1.3.2 Oblast zájmu

- Asymetrická změna obou stran s čímkoliv
- Změna kvality v důsledku zhoršení komprese s čímkoliv
- Logo nebo vodotisk v kombinaci s předcházejícími

1.4 Výběr reprezentativního vzorku

Pro skupinu vzájemně si podobných fotografií vybereme nejvhodnějšího kandidáta, který bude následně ostatní fotografie zastupovat. Jde o experimentální postup.

1.4.1 Návrh řešení

Bude zaveden koeficient zastupitelnosti (dále jen KoZ), který je zjednodušeně určen jako $VELIKOST * OSTROST + JAS$. Přičemž platí, že vyšší hodnota koeficientu zastupitelnosti znamená kvalitnější fotografiю (nikoliv na oko hezčí fotografiю). Reprezentativní vzorek bude fotografie, která bude vybraná ze skupiny podobných fotografií na základě KoP, s nejvyšším KoZ.

1.4.2 Oblast zájmu

Střední hodnota jasu je určena pomocí světlostní matice. Jedná se pouze o zohlednění, zda fotografie není příliš jasná nebo tmavá. Jde o jednoduchý algoritmus. Na výsledek nemá zásadní vliv.

Poměrný počet hran slouží jako test rozmazanosti fotografie. K realizaci se používá konvoluční matice (více v samostatné kapitole věnované této problematice) s vhodným jádrem typu horní i dolní propust (s celkovým součtem 0). Na výsledek má největší dopad.

Rozlišení fotografie je bráno jako klasická velikost fotografie v px (větší \Leftrightarrow lepší).

2 Koeficient podobnosti dvou fotografií

Jde o základní ukazatel podobnosti dvou fotografií. KoP leží na intervalu $<0,1>$. Přičemž hodnoty blížící se 1 symbolizují podobné fotografie. Interval podobnosti byl na základě testovacích pokusů stanoven na $<0,07,1)$. Hodnota 1 znamená duplicitní fotografii. Byla z intervalu vyloučena, jelikož jsou fotografie nejprve unifikovány pomocí otisku MD5 [6]. Výpočet KoP provedeme v šesti krocích, z čehož jsou čtyři kroky přípravné (optimalizační) a pouze dva kroky reálně ovlivňují výsledný KoP.

1. Změna velikosti fotografie
2. Převod na světlostní matici
3. Převod do vlnového spektra
4. Křížová korelace (Cross correlation method)
5. Převod zpět z vlnového spektra
6. Výpočet KoP z výsledné matice

Body 1–3 slouží jako přípravné a aplikují se na obě fotografie (vzorová a referenční). Do bodu 4 tedy vstupují dvě matice (pro každou fotografiю jedna). Výstupem 4. bodu je již jen jedna matice, která je v bodě 6 vyhodnocena do výsledného KoP.

2.1 Změna velikosti fotografie

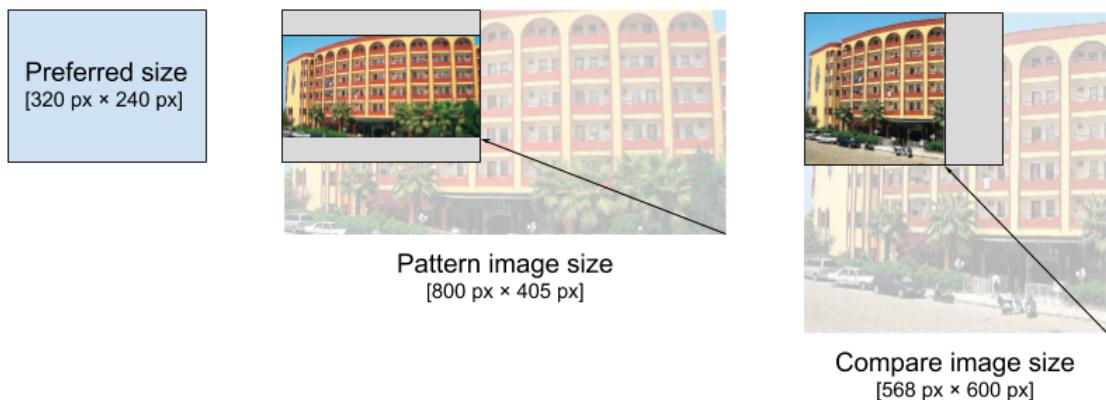
Jedná se o nezbytný přípravný krok, jehož cílem je sjednotit u obou fotografií počet bodů a tím také počet prvků v maticích, které vzniknou v následujícím kroku výpočtu KoP. Jako referenční velikost byla stanovena

- Šířka: 320 px,
- Výška: 240 px,

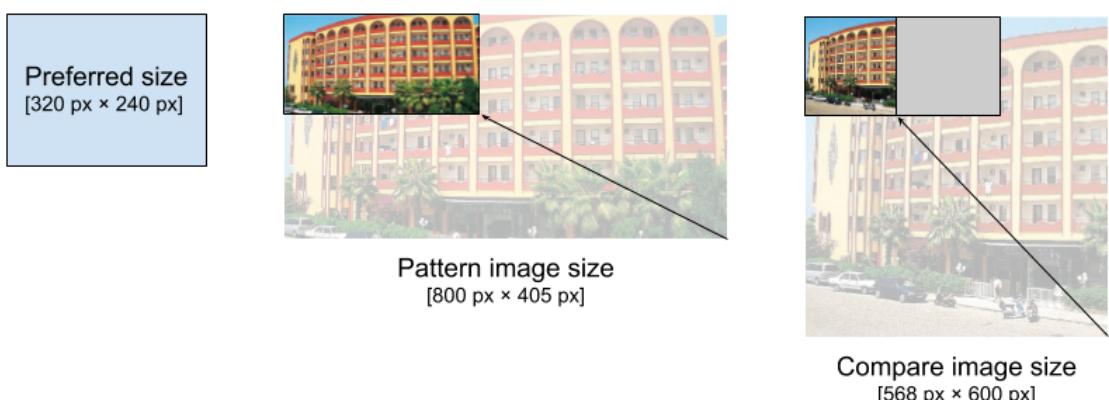
které jsou nevhodnějším kompromisem mezi relevancí výsledku a HW časem nutným k jeho zpracování.

Změna velikosti vzorové fotografie na $[320 \times ?]$ nebo $[? \times 240]$ se provádí v závislosti na delší straně. Kratší strana je dopočítána podle původního poměru stran. Výsledný rozměr není doplněn nulami na plnou referenční velikost $[320 \times 240]$. Vzorová fotografie tedy určuje velikost, na kterou musíme upravit referenční fotografii. Pokud má referenční fotografie jiný poměr stran, bude doplněna nulami, aby nevznikala prázdná místa.

Takto připravené fotografie nejsou při dalším zpracování komutativní, pokud mají vzájemně jiný poměr stran. V důsledku to znamená, že se musí porovnat fotografie v obou směrech (jak vzorová vůči referenční, tak referenční vůči vzorové). Situaci ukazují příklady na Obr. 2.1 a Obr. 2.2.



Obr. 2.1 Statický scaling obrázků (komutativní)



Obr. 2.2 Dynamický scaling obrázků (diskomutativní)

Přesto, že je tento proces HW dražší, než jeho komutativní varianta, získáme díky tomu řádově lepší relevanci výsledků (zejména pokud je jedna ze dvou fotografií velmi nekvalitní, případně má nižší nativní rozlišení, než je referenční).

2.2 Převod na světlostní matici

Světlostní matice představuje fotografii ve formátu rastrové bitmapy (někdy též šedotónový obraz; v anglicky psaných textech k nalezení pod názvem brightness-matrix). Formálně je to dvourozměrná diskrétní veličina, reprezentovaná maticí druhého rádu [7]. Každý bod původní fotografie - pixel [8] (dále jen px) je z RGB [9] hodnot převeden na hodnotu intenzity jasové funkce.

Původní px je reprezentován jako tří prvkové pole s hodnotami na intervalu $<0,255>$.

- R => red,
- G => green,

- $B \Rightarrow \text{blue}$,

Výsledná hodnota intenzity jasové funkce px se spočítá jako střední hodnota z hodnot jednotlivých složek px (R, G a B). Jak napovídá interval možných hodnot, jeden px převedený na prvek světlostní matici zabírá v operační paměti 1 Byte (1 Byte = 8 bit \Rightarrow osmibitová barevná hloubka \Rightarrow 256 stupňů šedi). Na jedenu plnou referenční fotografii je tedy potřeba 75 kB.

2.3 Převod do-z vlnového spektra

Převodem do vlnového spektra dosáhneme zajímavé výkonnostní optimalizace [10]. Kdybychom tento krok z celého procesu výpočtu KoP vynechali (stejně jako následně nezbytný převod zpět z vlnového spektra), museli bychom udělat křížovou korelací v normálním spektru. Tzn. $O(n^4)$ operací, kde n je velikost strany čtvercové matice.

Pokud ale nejprve provedeme Diskrétní Fourierovu transformaci (DFT) pro převod do vlnového spektra, až následně křížovou korelací a nakonec Zpětnou Fourierovu transformaci (IFT), ušetříme jeden řád hodnosti počtu operací. Budeme potřebovat $O(n^3 * \log n)$ operací. To je za předpokladu matice reprezentující fotografii 200×200 px rozdíl dvou řádů operací.

- Klasické spektrum: $200^4 = 1.6 * 10^9$ operací
- Vlnové spektrum: $200^3 * \log 200 = 6.1 * 10^7$ operací

2.4 Křížová korelace

Korelace je nejdůležitější krok celého výpočtu KoP. Umožní pomocí jednoduchých matematických operací rozhodnout, zda jsou světlostní matice vzorové a referenční fotografie podobné. Míra podobnosti je vyjádřena hodnotami korelačních koeficientů sestavených do jedné matice (zatím ještě ve vlnovém spektru). Čím více se výsledné hodnoty blíží 1, tím více si jsou fotografie podobné v daném bodě.

Celý proces není citlivý na konstruktivní změny. Dokáže tedy podobnost vyhodnotit bez ohledu na změnu sytosti barev, kontrastu či jasu. Stejně tak výsledek není ovlivněn přidáním vodotisku nebo šumem ve fotografii. Přidání loga do fotografie již sice sníží KoP, ale pokud není logo přes polovinu fotografie, je stále bezpečně rozpoznána jako podobná či nikoliv.

Formálně je korelace zejména statistický pojem, který označuje vzájemný lineární vztah mezi znaky nebo veličinami. Míra korelace je daná korelačním koeficientem [11].
Vzorec pro výpočet: $\rho_{A,B} = \frac{(A - \mu_A) \times (B - \mu_B)}{\sigma_A \sigma_B}$

Na první pohled se může zdát složitý, ale skrývá v sobě tři jednoduché ale důležité kroky.

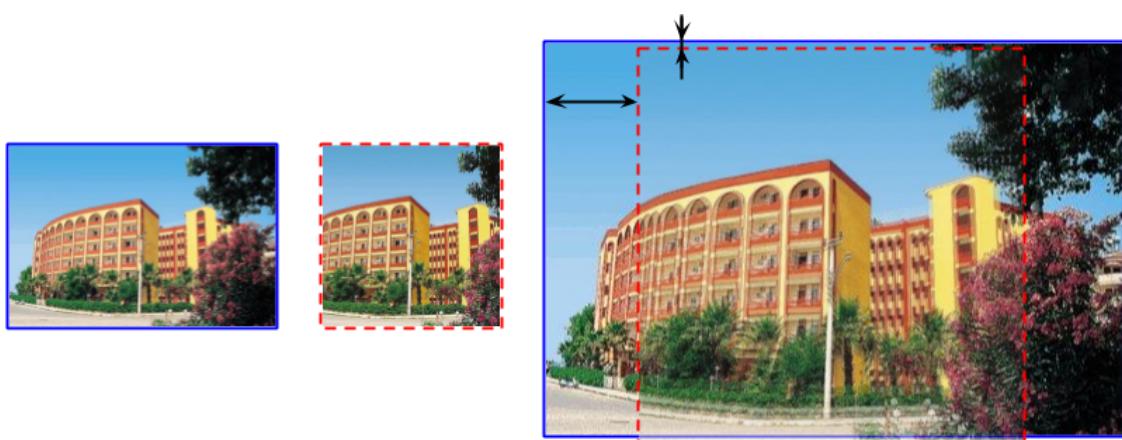
1. K maticím A a B se spočítá rozptyl σ_A, σ_B a střední hodnota μ_A, μ_B .
2. Odečtením střední hodnoty od matice $A - \mu_A, B - \mu_B$ je dosaženo **invariace nastavení jasu**.
3. Dělením rozptyly $\sigma_A\sigma_B$ je zajištěna **invariace nastavení kontrastu**.

Za předpokladu, že objekty na porovnávaných fotografiích nejsou vůči sobě v prostoru posunuty, poskytuje korelační matice odpověď na otázku, zda jsou si dvě fotografie podobné. V praxi ale tento model příliš nenastavá. Naopak je velmi časné, že je porovnáván např. výřez z fotografie oproti originálu, případně posunuté fotografie po horizontální či vertikální ose.

Tuto problematiku řeší křížová korelace (Cross correlation method) [12]. Princip samotné korelace je stejný, pouze se opakuje s částečným posunem tak, aby pokryla všechny možné kombinace mezi dvěma fotografiemi. Příklad je názorně vidět na původních fotografiích, jak ukazuje (Obr. 2.3) a (Obr. 2.4).



Obr. 2.3 Příklad proložení dvou fotografií s použitím klasické korelace



Obr. 2.4 Příklad proložení dvou fotografií s použitím křížové korelace

2.5 Výpočet výsledného koeficientu

Po převodu z vlnového spektra zpět vychází již matice, ze které jsou vyloučeny imaginární hodnoty a pracujeme tedy jen s reálnými čísly. Výsledný KoP pro dané fotografie je dán nejvyšším nalezeným korelačním koeficientem v matici.

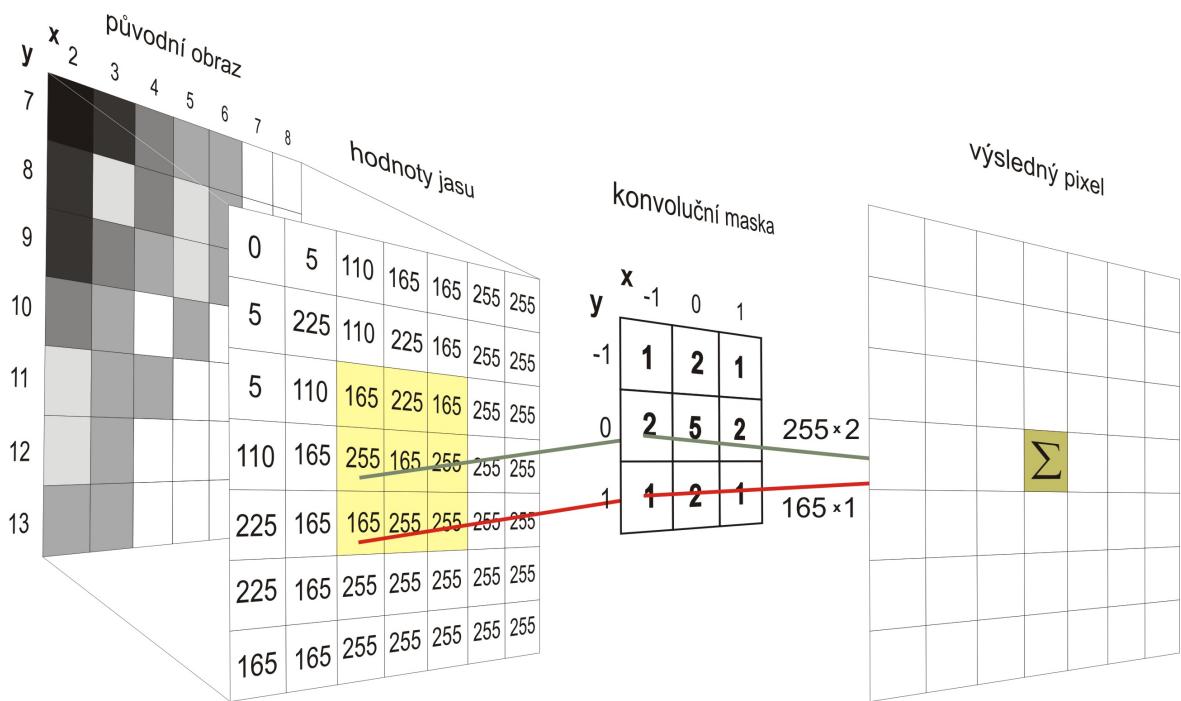
3 Koeficient zastupitelnosti dvou fotografií

Určení KoZ je experimentální proces založený na opakování aplikaci konvoluční matic s různým jádrem (konvoluční maskou). Bude jí věnována větší pozornost v projektové části. Zde budou představeny pouze techniky nutné k jejímu dosažení. Jelikož převod na světlostní matici a převod z–do vlnového spektra byly již představeny, nebudou dále znova uvedeny.

3.1 Diskrétní 2D konvoluce

Konvoluce [19] je matematický operátor pro zpracování dvou funkcí. V algoritmech zpracovávajících dvourozměrný diskrétní obraz (např. v počítačové grafice) má konvoluce následující tvar: $(f * h)(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f(x-i, y-j) \bullet h(i, j)$

Názorná ukázka konvoluce je vidět na přiloženém obrázku (Obr. 3.1).



Obr. 3.1 Příklad Diskrétní 2D konvoluce [19]

3.2 Konvoluce a volba jádra

Konvoluce ve světě počítačové grafiky je operace s obrázkem ve formátu matice pomocí jiné matice zvané „jádro“ (nebo též konvoluční maska). Jako první matice se používá obrázek určený k úpravě. Obrázek představuje dvojrozměrnou pravoúhlou souřadnicovou síť pixelů. Použité jádro závisí na požadovaném efektu.

V našem případě je požadovaný filtr na detekci hran [20]. Základní jádra pro detekci hran mají součet roven nule (Tab. 3.1 a Tab. 3.2). Nejsou vhodné na obrázky zatížené šumem. Ten je nezbytné před aplikací těchto filtrů eliminovat.

Tab. 3.1 Jádro pro detekci hran (horizontálně a vertikálně)

0	1	0
1	-4	1
0	1	0

Tab. 3.2 Jádro pro detekci hran (horizontálně, vertikálně a šikmé hrany)

1	1	1
1	-8	1
1	1	1

II. ANALYTICKÁ ČÁST

4 Brainstorming

Tato analytická metoda slouží ke sběru myšlenek, námětů a případné zevrubné konstruktivní kritice dané problematiky. V rámci této práce byla použita v akademickém a profesním kruhu za účelem identifikace základních ukazatelů pro další kroky analýzy.

4.1 Akademický kruh

Diskutovány byly zejména technické možnosti týkající se otázek kde a jak lze vůbec porovnání fotografií provádět strojově. K další analýze byly vyb

4.2 Profesní kruh

V kruhu s provozovatelem byly kladený nejvyšší nároky na flexibilitu a propustnost celého řešení.

5 Diferenční analýza (GAP analýza)

5.1 Popis současného stavu

Je požadováno porovnání rastrových bitmap za účelem identifikace vzájemně podobných fotografií. Řešení je hledáno pro produkční provoz. Konzumentem cílového řešení je webový portál dovolena.cz, který má přibližně dva miliony fotografií. Průměrný počet přístupů k některé fotografii je přibližně 100 přístupů za sekundu. Obsahem fotografií jsou především hotely a jejich okolí. Webový portál slouží spíše jako datový konsolidátor. Nabídka portálu značně ovlivňuje cílové portfolio fotografií. Podle testovacích měření se za jeden týden obmění cca 10% fotografií z celkového množství. Jako testovací vzorek byl vybrán jeden nejmenovaný hotel a jeho 35 fotografií. Zákazník webového portálu vidí všechny fotografie v nesetříděné galerii. Některé fotografie jsou unikátní, ale většina si je velmi podobná. U některých dokonce nejsou lidským okem patrné rozdíly.

5.2 Popis cílového stavu

Konsolidované fotografie prezentované klientovi budou v maximální možné míře obsahovat unikátní fotografie. Vzájemně si podobné fotografie budou odfiltrovány a zůstane pouze jedna a to fotografie s nejvyšším KoZ. Klient nebude čekat na zpracování podobnosti obrázků. Pokud budou zpracované, klient uvidí jen unikáty. Pokud nebudou zpracované, klient uvidí vše v původním stavu. V takovém případě se poměrově zvýší priorita na výpočet podobnosti fotografií tohoto hotelu vůči ostatním ve frontě na výpočet. Cílové řešení musí být schopno operovat řádově s jednotkami milionů fotografií s týdenní fluktuací 15%.

5.2.1 Nefunkční požadavky

- Bezúdržbový systém
- Nevyžadující v průběhu času další financování
- Minimální vstupní investice
- Maximální kompatibilita s aktuálním HW
- Programovací jazyk Java [24]

5.3 Rozdíly

- Nově vznikne nástroj pro určení KoP.
- Nově vznikne nástroj pro určení KoZ.
- Fotografie jednoho hotelu budou oindexovány a vnitřně škálovány do skupin pomocí koeficientů výše.
- Dojde k navýšení celkového počtu fotografií.
- Zvýší se fluktuace fotografií (na očekávaných 15%).

5.4 Návrh variant k dosažení cíle

Pilířem celého řešení bude backendová strana cílového konzumenta. Limity a také jednotlivé možnosti pro realizaci jsou velmi omezeny nutností integrovat do současného řešení. Z těchto důvodů má serverová strana převážně podpůrný charakter v projektu jako celku. Její význam je zejména v propojení všech jednotlivých komponent. Dojde tedy k modifikaci existujícího produkčního server-side prostředí. Nově zde poběží služba, která bude

- poskytovat zadání na určení KoP,
- poskytovat metadata nezbytná pro distribuci výpočtu,
- konzumovat výsledek distribuované operace,
- kompletovat zpracovaná data do cache vhodné pro silný organický provoz.

Naopak klientská strana je naprosto autonomní. Pro realizaci lze použít jak libovolnou platformu, tak libovolné technologie. Jediným technickým limitem je schopnost standardizovaným způsobem komunikovat se serverovou stranou pomocí SOAP api [21].

5.4.1 Výpočet KoP a KoZ na CPU produkčního serveru

Základní myšlenka je využít nejdostupnější produkční HW a na zavedeném serveru spustit novou službu. Hlavní výhodou je dostupnost produkčního HW ve vlastním datacentru cílového konzumenta. Podstatnou nevýhodou je fakt, že pro určení výše uvedených koeficientů není CPU [22] ideální platforma.

5.4.2 Výpočet KoP a KoZ na CPU produkčního serveru s paralelizací na GPU

Tento způsob předpokládá osazení produkčního serveru dedikovanou pracovní grafickou kartou a vybrané části procesu výpočtu KoP a KoZ optimalizovat pro zpracování na GPU [23].

5.4.3 Výpočet koeficientů na PC farmě

Způsob předpokládá využití HW osobních PC, kterých je v každé větší společnosti dostatek. Výpočet koeficientů tedy není nutno provádět na jednom stroji, ale lze jej provádět na každém dostupném stroji.

5.5 Zhodnocení variant

Jako nejvhodnější varianta pro realizaci vychází PC farma. Jako záložní řešení lze využít produkční server s výpočtem na CPU. Pokud bude zvolen vhodný programovací jazyk (např. požadovaný programovací jazyk Java [24]), bude výsledné řešení přenositelné.

Zhodnocení výkonových rozdílů vychází z benchmarkingu, který byl vyhodnocen jako pomocná analýza uvedená níže. Zhodnocení prezentuje tabulka (Tab. 5.1)

Tab. 5.1 Tabulka výsledků jednotlivých variant krátkodobého testu

Varianta	Týdenní přírůstek	Všechny fotografie	Teoretická rezerva
CPU	6 dní	27 týdnů	3,7%
GPU	8 hodin	2 dny 2 hodiny	300%
PC farma (300 ks)	2,5 hodin	16 hodin	700%

Zhodnocení investičních rozdílů ukazuje tabulka (Tab. 5.2). Vychází z předpokladu, že x je investiční konstanta v Kč, vztažená na cenu jednoho produkčního serveru bez GPU. Pro zjednodušení je stanovena na 100.000 Kč (není příliš daleko od reality).

Tab. 5.2 Tabulka výsledků jednotlivých variant krátkodobého testu

Varianta	Pořizovací cena [Kč]	Nutná investice [Kč]
CPU	x	0
GPU	$10x$	$10x$
PC farma (300 ks)	$30x$	0

5.5.1 Výpočet KoP a KoZ na CPU produkčního serveru

Výsledky banchmarkingu

- Za necelých 6 dní spočítá týdenní přírůstek.
- Za zbylou dobu z týdenního cyklu dále vypočítá přibližně 3,7% z celkového objemu sto milionu koeficientů.
- Pro plné vypočítání všech koeficientů potřebuje dalších cca 27 týdnů.
- Rezerva pro další růst (navýšení celkového počtu fotografií) je cca 18%.

5.5.2 Výpočet KoP a KoZ na CPU produkčního serveru s paralelizací na GPU

Pro realizaci tohoto scénáře je nezbytná vysoká vstupní investice. Běžně dostupný produkční server nedisponuje GPU. Pořizovací cena takového serveru je řádově vyšší. Dále je nutné připočítat cenu vlastní dedikované grafické karty, která cenově převyšuje cenu serveru. Její výběr je velmi omezen kompatibilitou s produkčními servery. Zástupce firmy Dell byl schopen nacenit na vlastní server pouze dvě karty s možností garancí non-stop provozu a výměnou do 24 hodin.

Výsledky banchmarkingu

- Za necelých 8 hodin spočítá týdenní přírůstek.

- Za část ze zbylé doby týdenního cyklu dále vypočítá 100% z celkového objemu sto milionu koeficientů.
- Pro plné vypočítání všech koeficientů potřebuje celkem 2 dny.
- Rezerva pro další růst (navýšení celkového počtu fotografií) je cca 300%.

5.5.3 Výpočet koeficientů na PC farmě

Výsledek jednoho kancelářského PC je zanedbatelný a nemůže se rovnat předchozím variantám. Vezmeme-li v úvahu, že těchto strojů je k dispozici 300 kusů po dobu 16 hodin denně, posunou se výsledky na jinou úroveň.

Výsledky banchmarkingu

- Za 2,5 hodiny spočítá týdenní přírůstek (za předpokladu 300 aktivních PC).
- Za část ze zbylé doby týdenního cyklu dále vypočítá 100% z celkového objemu sto milionu koeficientů.
- Pro plné vypočítání všech koeficientů potřebuje celkem 16 hodin, což je v tomto případě 1 den.
- Rezerva pro další růst (navýšení celkového počtu fotografií) je cca 700%.

Tento krok předpokládá mimo jiné i revizi infrastruktury, především kvalitu a šířku pásmá sítě, centrální správu PC apod., které v této fázi nejsou zahrnuty do hodnotících kritérií.

6 Benchmarking

Jednotlivé kandidáty pro realizaci klientské části (zavedené v diferenční analýze) podrobíme výkonnostním a srovnávacím testům. Základní předpoklady:

- Řádově je potřeba jednorázově odbavit 100.000.000 výpočtů KoP.
- Řádově je potřeba jednorázově odbavit 2.000.000 výpočtů KoZ.
- Na týdenní bázi následně odbavovat přírůstky řádově
 - 15.000.000 výpočtů KoP,
 - 300.000 výpočtů KoZ.

6.1 Testovací prostředí

6.1.1 Použitý HW

Simulace produkčního serveru je založena na pracovní stanici Lenovo D20, která je svoji sestavou velmi blízko produkčnímu serveru. Vzhledem k tomu, že disponuje starší generací CPU, bylo do ní osazeno i odpovídají GPU, aby bylo možné výsledky approximovat na reálný produkční HW.

- Lenovo ThinkStation D20
- Operační systém Windows 10 pro 64 bit
- CPU Intel(R) Xeon(R) X5670 @ 2.93 GHz (2×cpu, 6×core, 12×thread)
- RAM 32 GB DDR3 (1066)
- GPU NVIDIA GeForce GTX 460
- HDD SAS 10.000 ot

Simulace klasického PC pro je založena na PC Dell Optiplex 780. Obecně je to velmi rozšířený kancelářský PC. Taktéž cílový konzument disponuje více než 300 ks odpovídajících kancelářských PC. Tato simulace je omezena pouze na výpočty na CPU.

- Dell Optiplex 780
- Operační systém Windows 7 pro 32 bit
- CPU Intel Core 2 Duo E7500 2,93 GHz
- RAM 4 GB DDR3
- HDD SATA 250 GB

6.1.2 Použitý SW

V úvodní fázi projektu probíhal vývoj i testování algoritmů v Matlabu [4]. Ten vnitřně používá pro zpracování DFT a IFT knihovnu FFTW [13]. Tato knihovna má dostupnou nativní implementaci v CMake [17]. Dále má pro řadu programovacích jazyků (včetně programovacího jazyku Java [16]) připravený wrapper [18].

Algoritmu v matlabu (výpočet KoP na CPU) níže (Kod. 6.1) byl použit jako základní srovnávací test pro výpočet KoP hrubou silou na CPU. Výsledný KoP odpovídá v algoritmu proměnné ccCMax.

Kod. 6.1 banchmark-cpu.m

```

1 close all; clc; clear;
2
3 method = 'nearest';
4 %method = 'bicubic';
5
6 type = 'single';
7
8 n = 35; % pocet obrazku
9
10 %obrazky
11 im_gpu = cell(1, n);
12
13 h = zeros(1, n, type);
14 w = zeros(1, n, type);
15
16 s0 = 'hotel_images\aa';
17
18 %nahravani obrazku
19 tic
20 for i = 1:n
21     if i < 10
22         s = [s0, '0' int2str(i), '.tif'];
23     else
24         s = [s0,int2str(i),'.tif'];
25     end;
26
27     im = single(imread(s)); %nahrani obrazku a prevod
28
29 [n1,n2,n3] = size(im);
30 h(i) = n1;
31 w(i) = n2;
32 im_gpu{i} = im; %nahrani obrazku
33 end;
34 disp('Nahravani vsech obrazku:');
35 toc
36
37 %prevedeni obrazku na jasove matice
38 for i=1:n
39     im_gpu{i} = sum(im_gpu{i}, 3) ./ 3; %preved na jasovou matici
40 end;
41 disp('Prevod vsech obrazku na jasovou matici:');
42 toc
43
44 log_cpu = zeros(n, n, type);
45
46 %cyklus
47 for i=1:n
48     %ft prvniho obrazku
49     ft1 = fft2(im_gpu{i});
50     ft1Norm = ft1 ./ abs(ft1);
51
52     for j=1:n
53         if i==j
54             continue;
55         end;
56
57         scale = min(h(i)/h(j), w(i)/w(j));
58         h2 = round(scale * h(j));
59         h2 = min(h2, h(i));
60         w2 = round(scale * w(j));
61         w2 = min(w2, w(i));
62
63         im2Sc = imresize(im_gpu{j}, [h2, w2], method);
64
65         %doplneni druheho obrazku nulami na jednotnu velikost
66         im2 = zeros(h(i), w(i), type);
67         im2(1:h2, 1:w2) = im2Sc;
68
69         %ft druheho obrazku
70         ft2 = fft2(im2);
71         ft2Norm = ft2 ./ abs(ft2);
72
73         skrizova korelace ve vlnovem spektru
74         ccW = ft1Norm .* conj(ft2Norm);
75
76         %preved do obycejneho souradnicoveho systemu
77         ccC = real(ifft2(ccW));
78
79         %shledani maximalniho korelacniho koeficientu
80         ccCMax = max(ccC);
81         ccCMax = max(ccCMax);
82         log_cpu(i,j) = ccCMax;
83         disp(['obr1: ',int2str(i),'; obr2: ',int2str(j)]);
84     end;
85 end;
86 disp(['Cyklus ',int2str(n),' obrazku:']);
87 time = toc
88
89 clearvars -except log_cpu;

```

Algoritmus v matlabu (výpočet KoP s paralelizací na GPU) níže (Kod. 6.2) vychází z původního zpracování (Kod. 6.1). Vybrané části výpočtu KoP jsou zpracovány pomocí paralelizace na GPU. Výsledný KoP opět odpovídá proměnné ccCMax. Pro paralelizaci byla využita platforma CUDA [25].

Kod. 6.2 banchmark-gpu.m

```

1 %setenv('NSIGHT_CUDA_DEBUGGER','1')
2 close all; clear; clc;
3
4 %method = '_nearest';
5 %method = '_bicquadrat';
6 method = '_bicubic';
7
8 type = 'single';
9
10 n = 35;
11
12 path = 'kernels\ImgScale\ImgScale\kernel_';
13 kernel = parallel.gpu.CUDAKernel([path, type, method, '.ptx'], [path, type, method, '.cu']);
14
15 %obrazky
16 im_gpu = cell(1,n);
17
18 h = zeros(1, n, type);
19 w = zeros(1, n, type);
20
21 s0 = 'hotel_images\aa';
22
23 %nahravani obrazku na grafiku
24 tic
25 for i = 1:n
26     if i < 10
27         s = [s0,'0',int2str(i),'.tif'];
28     else
29         s = [s0,int2str(i),'.tif'];
30     end
31
32     im = single(imread(s)); %nahrani obrazku a prevod
33
34     [n1,n2,n3] = size(im);
35     h(i) = n1;
36     w(i) = n2;
37     im_gpu{i} = gpuArray(im); %nahrani obrazku na grafiku
38
39 disp('Nahravani vsech obrazku do pameti grafiky:');
40 toc
41
42 %prevadeni obrazku na jasove matici
43 for i=1:n
44     im_gpu{i} = sum(im_gpu{i}, 3) ./ 3; %prevod na jasovou matici
45 end;
46 disp('Prevod vsech obrazku na jasovou matici:');
47 toc
48
49 log = zeros(n, n, type, 'gpuArray');
50
51 %cyklus
52 for i=1:n
53     %ft prvniho obrazku
54     ft1 = fft2(im_gpu{i});
55     ft1Norm = ft1 ./ abs(ft1);
56
57     for j=1:n
58         if i==j
59             continue;
60         end;
61
62         scale = (min(h(i)/h(j), w(i)/w(j)));
63         h2 = round(scale * h(j));
64         h2 = min(h2, h(i));
65         w2 = round(scale * w(j));
66         w2 = min(w2, w(i));
67         scale_h = h(j)/h2;
68         scale_w = w(j)/w2;
69
70         %im2Sc = imresize(im_gpu{j}, scale);
71
72         im2Sc = zeros(h2, w2, type, 'gpuArray');
73         kernel.GridSize = w2;
74         kernel.ThreadBlockSize = h2;
75         im2Sc = feval(kernel, im2Sc, im_gpu{j}, int32(h(j)), int32(w(j)), scale_h, scale_w);
76
77         %doplneni druhoho obrazku nulami na jednotnou velikost
78         im2 = zeros(h(i), w(i), type, 'gpuArray');
79         im2(1:h2, 1:w2) = im2Sc;
80
81         %ft druhoho obrazku
82         ft2 = fft2(im2);
83         ft2Norm = ft2 ./ abs(ft2);
84
85         %skrizova korelace ve frekvencnim spektru
86         ccW = ft1Norm .* conj(ft2Norm);
87
88         %sprechod do obycejneho souradnicoveho systemu
89         ccC = real(ifft2(ccW));
90
91         %shledani maximalniho korelacniho koeficientu
92         ccCMax = max(ccC);
93         ccCMax = max(ccCMax);
94         log(i,j) = ccCMax;
95         disp(['obr1: ',int2str(i),'; obr2: ',int2str(j)]);
96     end
97 end
98 disp(['Cyklus ',int2str(n),' obrazku:']);
99 log_gpu = gather(log);
100 toc
101
102 clearvars -except log_gpu;

```

Algoritmus v matlabu (výpočet KoZ na CPU) níže (Kod. 6.3) je svou strukturou a celkovým provedením odlišný od předchozích algoritmů (Kod. 6.1 a Kod. 6.2). Některé prvky (např. výpočet světlostní matici) zůstaly zachovány. Výpočet KoZ byl po provedení testů ve srovnávacích testech zanedbán. Především proto, že se nemusí počítat

pro kombinace fotografií, ale právě jednou pro každou fotografií a to při jejím pořízení. Průměrná doba výpočtu KoZ je 620 ms.

Kod. 6.3 banchmark-koz-cpu.m

```

1 a = cell(1,15);
2   a{7} = 'image/006780_5305124.jpg';
3 a{10} = 'image/37342_cat@008802_069_01.jpg';
4 a{13} = 'image/37342_cat@09629_110_10.jpg';
5 a{4} = 'image/37342_010590_11837774.jpg';
6 a{11} = 'image/37342_cat@08103_113_02.jpg';
7 a{1} = 'image/37342_cat@09015_115_06.jpg';
8 a{9} = 'image/37342_cat@11203_140_04.jpg';
9 a{6} = 'image/cat@013326_136_04.jpg';
10 a{14} = 'image/cat@012556_124_06.jpg';
11 a{5} = 'image/37342_cat@10488_142_06.jpg';
12 a{15} = 'image/cat@013842_136_04.jpg';
13 a{8} = 'image/lbc402_zimmerl_0912.jpg';
14 a{12} = 'image/37342_cat@09533_052_02.jpg';
15 a{2} = 'image/37342_cat@11000_055_02.jpg';
16 a{3} = 'image/37342_cat@11855_060_02.jpg';
17
18 close all;
19
20 startfile = 1;
21 endfile = 15;
22
23 best = zeros(endfile - startfile+1,4);
24
25 kernel_up = [-0.0625 -0.0625 -0.0625; -0.0625 0.5 -0.0625; -0.0625 -0.0625 -0.0625];
26
27 kernel_down = [0 0.125 0; 0.125 0.5 0.125; 0 0.125 0];
28
29 for i=startfile:endfile
30   im = imread(a{i});
31   [m,n] = size(im(:,:,1));
32
33   im_br = sum(im,3)/3;
34   im_ft = fft2(im_br);
35
36   kernel_up_sized = zeros(m,n);
37   kernel_up_sized(1:2,1:2) = kernel_up(2:3,2:3);
38   kernel_up_sized(m,1:2) = kernel_up(1,2:3);
39   kernel_up_sized(1:2,n) = kernel_up(2:3,1);
40   kernel_up_sized(m,n) = kernel_up(1,1);
41   kernel_up_ft = fft2(kernel_up_sized);
42
43   im_conv1_ft = im_ft.*kernel_up_ft;
44   im_conv1 = ifft2(im_conv1_ft);
45
46   kernel_down_sized = zeros(m,n);
47   kernel_down_sized(1:2,1:2) = kernel_down(2:3,2:3);
48   kernel_down_sized(m,1:2) = kernel_down(1,2:3);
49   kernel_down_sized(1:2,n) = kernel_down(2:3,1);
50   kernel_down_sized(m,n) = kernel_down(1,1);
51   kernel_down_ft = fft2(kernel_down_sized);
52
53   im_conv2_ft = im_conv1_ft.*kernel_down_ft;
54   im_conv2 = ifft2(im_conv2_ft);
55
56   im_conv1_8 = uint8(zeros(m,n,3));
57   im_conv1_8(:,:,1) = uint8(im_conv1);
58   im_conv1_8(:,:,2) = uint8(im_conv1);
59   im_conv1_8(:,:,3) = uint8(im_conv1);
60   im_conv2_8 = uint8(zeros(m,n,3));
61   im_conv2_8(:,:,1) = uint8(im_conv2);
62   im_conv2_8(:,:,2) = uint8(im_conv2);
63   im_conv2_8(:,:,3) = uint8(im_conv2);
64
65 % if i==1 || i==2 % || i==9
66 %   figure;
67 %   image(im_conv1_8);
68 %   figure;
69 %   image(im_conv2_8);
70 %
71
72 sharpness = sum(sum(im_conv1 > 20));
73 oversharpeness = sum(sum(im_conv1-im_conv2 > 20));
74
75 best(i-startfile+1,1) = i;
76 best(i-startfile+1,2) = sharpness;
77 best(i-startfile+1,3) = oversharpeness;
78 best(i-startfile+1,4) = sharpness / max(250,oversharpeness-250);
79
80 end;
81
82 [sbest, sx] = sort(best(:,4),'descend');
83 best = best(sx,:);

```

6.2 Výsledky testování

Byly realizovány tři testovací scénáře.

- Výpočet čistě na CPU s maximálním využitím produkčního HW.
- Výpočet na CPU s využitím GPU pro paralelizaci vybraných procesů.
- Výpočet čistě na CPU na PC farmě.

6.2.1 Výpočet koeficientů na CPU

Výpočet na CPU byl spuštěn v pěti vláknech. Po spuštění probíhal výpočet velmi slibně. Bohužel při delším testu se výpočet začal značně propadat. Později se ukázalo, že hlavním důvodem je odložené uvolňování RAM. Jinými slovy dokud byla další volná RAM, výpočet jel velmi rychle. S potřebou uvolnit RAM se výpočet řádově snížil.

- Krátkodobý test
 - doba: 5 minut
 - počet opakování testu: 10
 - průměrně za sekundu: 205 výpočtů obou koeficientů
- Dlouhodobý test
 - doba: 6 hodin
 - počet opakování testu: 10
 - průměrně za sekundu: 31 výpočtů obou koeficientů
- Propad o 85%

Na obrázku níže (Obr. 6.1) je výstup z profilování výpočtu na CPU. Obsahuje poměrově zvýrazněné dlouho trvající operace vůči zbytku procesu. V rámci profilování bylo zpracováno deset tisíc iterací a doby jednotlivých částí zprůměrovány.



Obr. 6.1 Vizualizace procesu výpočtu koeficientů na CPU

Celková zátěž alokovaného HW byla oproti očekávání výrazně nižší. Nepodařilo se efektivně využít CPU, které dosahovalo průměrné zátěže 30%. Hlavní problém je re-alokace operační paměti. Jednotlivé procesy pak čekají na přidělení operační paměti. Úvodní myšlenka využít hrubou sílu produkčního HW se ukázala jako nevyhovující.

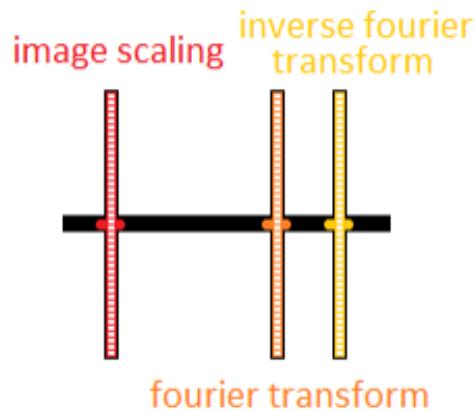
6.2.2 Výpočet koeficientů na CPU s paralelizací na GPU

Výpočet byl spuštěn v jednom vlákně na CPU. Vhodné operace pro GPU jsou delegovány pro paralelní zpracování. Nejprve byl na GPU paralelizován pouze úvodní scaling obrázků, což nemělo přijatelný dopad na výsledky. Následně byly paralelizovány na GPU také prováděné transformace obrázků. Právě tento krok měl zásadní vliv na zrychlení celé operace. Je opět patrný mírný propad krátkodobého testu oproti dlouhodobému.

- Krátkodobý test
 - doba: 5 minut

- počet opakování testu: 10
- průměrně za sekundu: 617 výpočtů obou koeficientů
- Dlouhodobý test
 - doba: 6 hodin
 - počet opakování testu: 10
 - průměrně za sekundu: 559 výpočtů obou koeficientů
- Propad o 10%

Na obrázku níže (Obr. 6.2) je výstup z profilování výpočtu na CPU s využitím GPU pro paralelizací dlouhotrvajících operací na CPU. Obsahuje poměrově zvýrazněné dlouho trvající operace vůči zbytku procesu. V rámci profilování bylo zpracováno deset tisíc iterací a doby jednotlivých částí zprůměrovány.



Obr. 6.2 Vizualizace procesu výpočtu koeficientů na CPU s paralelizací na GPU

Zátěž na CPU dosahuje v průměru 5%. Naopak GPU dosahuje zátěže cca 75%. Otázkou zůstává, jak dlouho je schopná GPU pracovat pod tímto permanentním zatížením.

6.2.3 Výpočet koeficientů na PC farmě

Výpočet byl spuštěn v jednom pracovním vlákně s omezenou možností alokace RAM na 1GB. Jedná se v podstatě o alternativu výpočtu obou koeficientů čistě na CPU. Tomu odpovídá i celkový průběh zpracování, který byl téměř stejný, pouze s nižší dotací vypočítaných dvojic koeficientů. Lze tedy přejít rovnou na výsledky, jelikož samotné zpracování nic nového nepřineslo.

- Krátkodobý test
 - doba: 5 minut

- počet opakování testu: 10
- průměrně za sekundu: 10 výpočtů obou koeficientů
- Dlouhodobý test
 - doba: 6 hodin
 - počet opakování testu: 10
 - průměrně za sekundu: 6 výpočtů obou koeficientů
- Propad o 40%

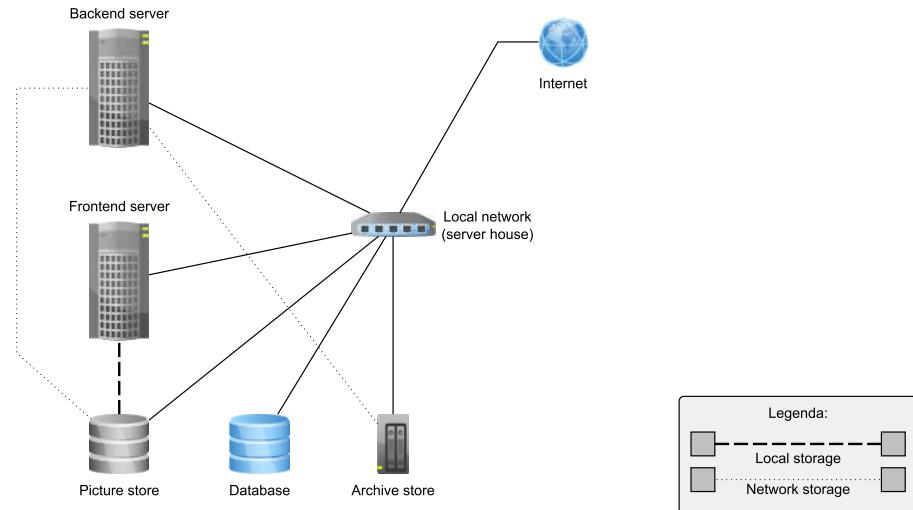
Zátěž na CPU je konstantní. Jedno jádro ze dvou jede permanentně na 100%. Celková zátěž CPU tedy 50%. Vzhledem k tomu, že stoj již netrpěl na realokaci paměti, je v případě PC úzkým hrdlem opravdu CPU. Ostatní sledované metriky (vytížení HDD, síťový provoz, pracovní teplota) se příliš nevychýlily z normálu.

III. PROJEKTOVÁ ČÁST

7 Příprava projektu

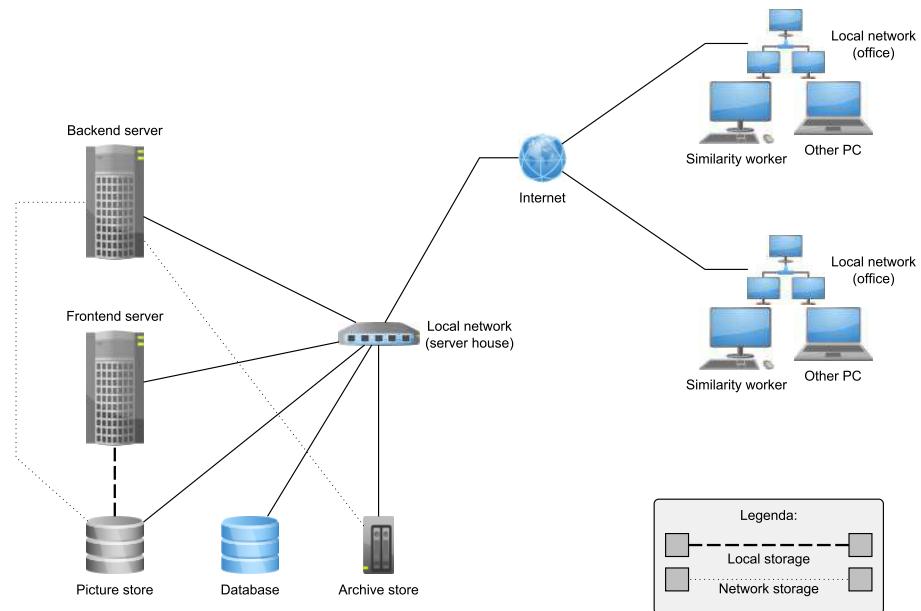
7.1 HW Architektura

7.1.1 Pohled na aktuální HW mapu



Obr. 7.1 L1 pohled aktuálního stavu

7.1.2 Pohled na plánovanou HW mapu

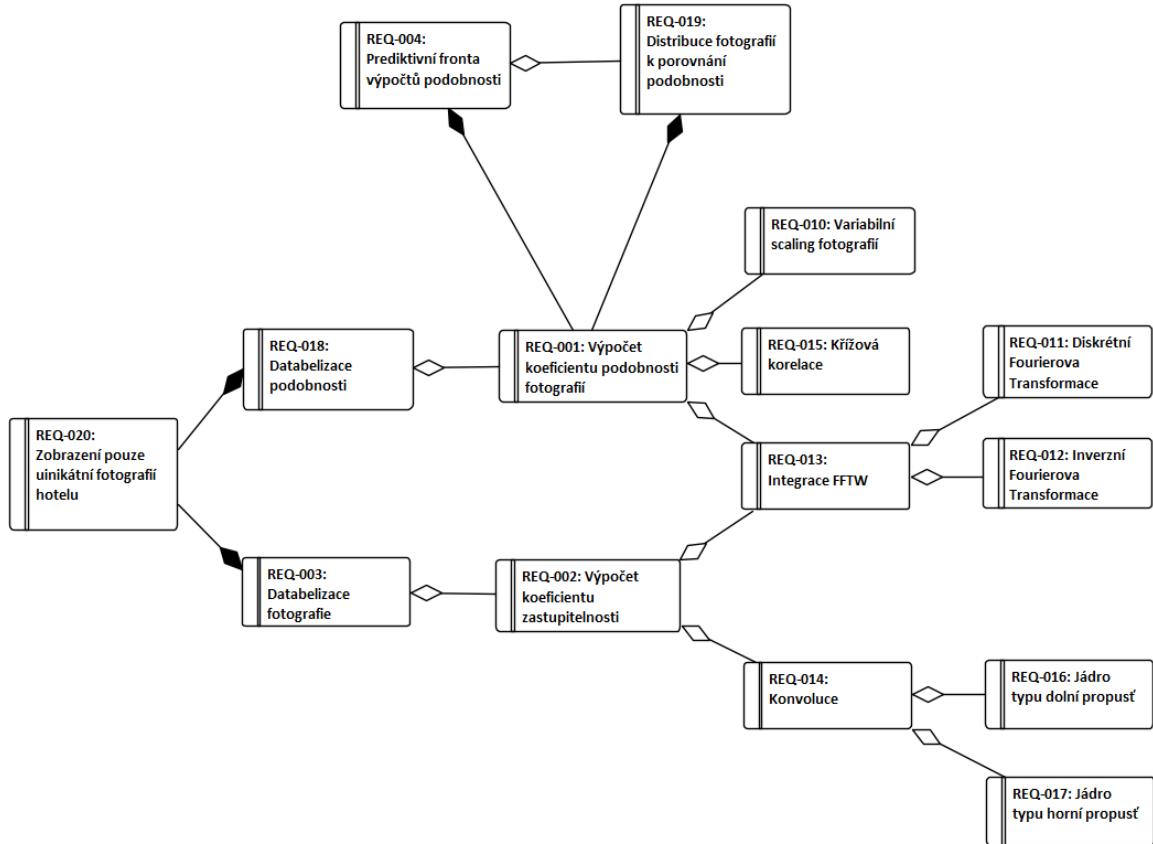


Obr. 7.2 L1 pohled cílového stavu

7.2 Požadavky

7.2.1 Funkční požadavky

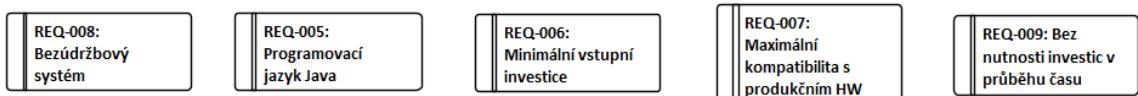
Klasifikaci funkčních požadavků systému znázorňuje obrázek (Obr. 7.3).



Obr. 7.3 Funkční požadavky

7.2.2 Nefunkční požadavky

Klasifikaci nefunkčních požadavků systému znázorňuje obrázek (Obr. 7.4).

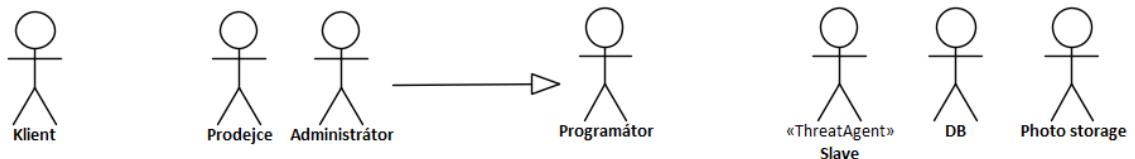


Obr. 7.4 Nefunkční požadavky

7.3 Případy použití

7.3.1 Aktéři

Klasifikace aktérů (Obr. 7.5).



Obr. 7.5 Aktéři

7.3.2 Případy užití

Klasifikaci případů použití znázorňuje obrázek (Obr. 7.6).

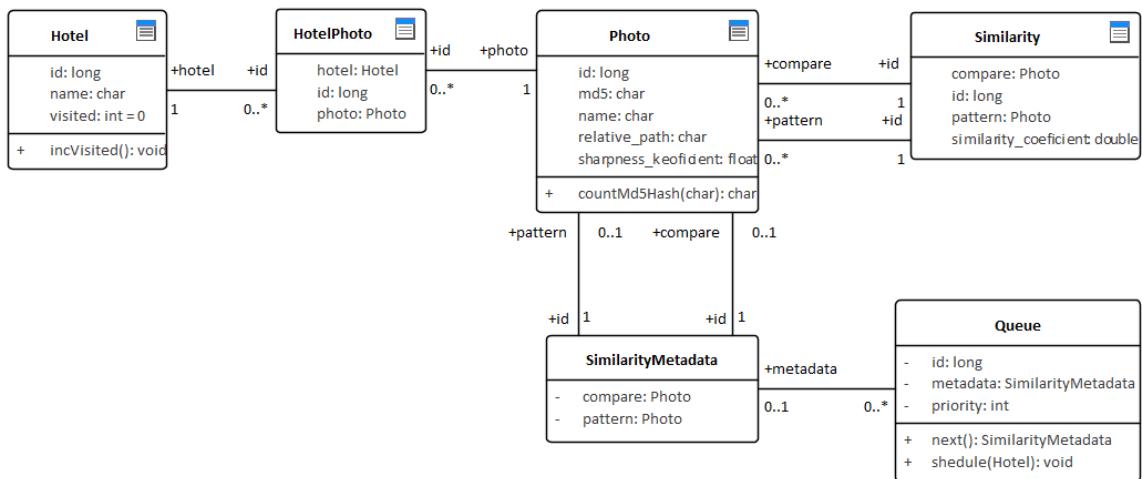


Obr. 7.6 Případy užití

7.4 Modely

7.4.1 Model tříd

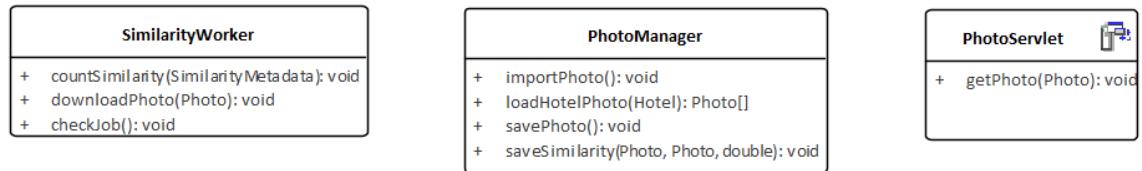
Identifikované datové modely jsou znázorněny pomocí modelu třídy na obrázku níže (Obr. 7.7)



Obr. 7.7 Model tříd

7.4.2 Model služeb

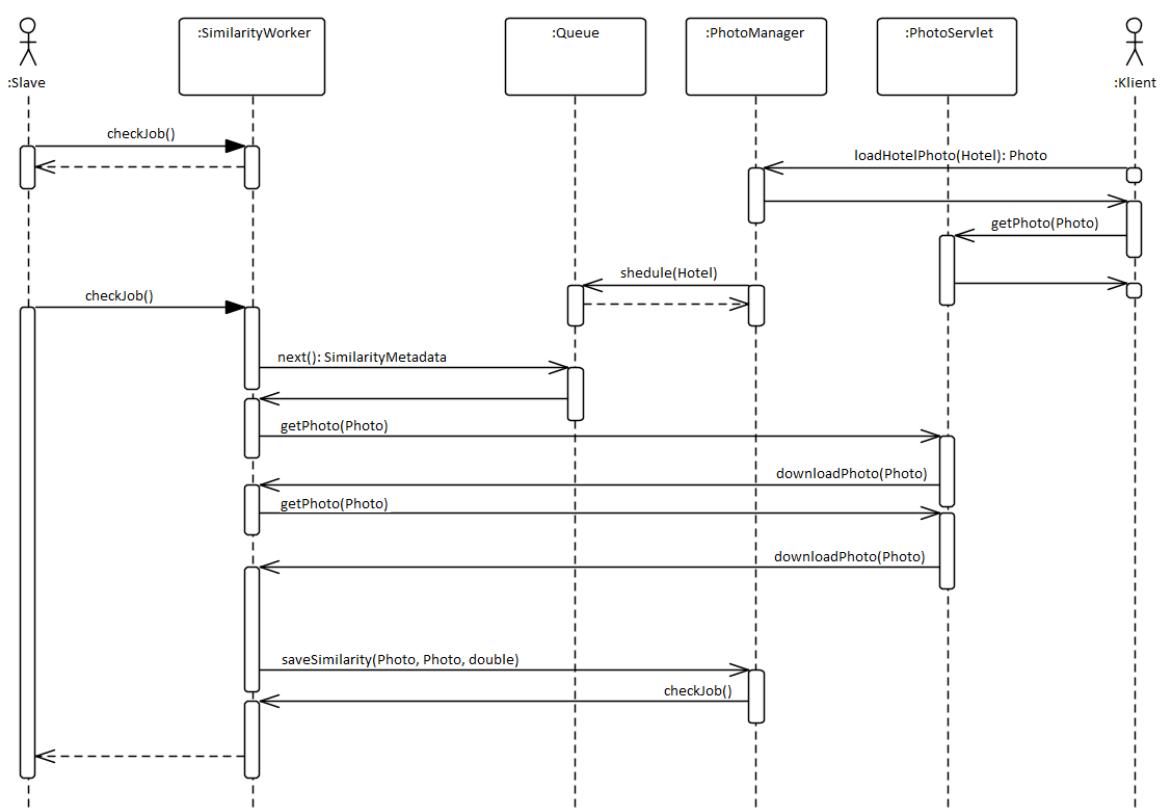
Identifikované backendové služby jsou znázorněny na obrázku níže (Obr. 7.8)



Obr. 7.8 Model služeb

7.5 Sekvenční diagram distribuované služby

Na obrázku níže (Obr. 7.9) je znázorněn sekvenční diagram přibližující odbavení výpočtu KoP.



Obr. 7.9 Sekvenční diagram

8 Serverová strana distribuované služby

Současná serverová strana je webová aplikace postavená na JavaSE 7 [26] a Springu [27]. Pro komunikaci využívá SOAP API. Data ukládá do relační databáze (PostgreSQL [29]). Tato serverová strana bude modifikována. Nově zde bude

- upraven proces databelializace fotografií (výpočet KoZ),
- zkonstruována fronta importovaných fotografií čekajících na výpočet KoP,
- vystaven servlet navrzející fotografie,
- vystavena webová služba pro obsluhu distribuované služby.

8.1 Databelializace fotografií

Do původního řešení byla zavedena kontrola existence totožné fotografie na základě MD5. Hodnota MD5 je ukládána do DB. Nad tímto sloupec je aplikován unikátní index. Dále se nově při importu fotografie ukládají její vybraná metadata. Ty později s výhodou používají na výpočet KoZ přímo v SQL dotazu. Jedná se o:

- výšku a šířku v px
- poměrný počet hran (přeostřenost)

Výsledek po dokončení importu fotografií je patrný na obrázku níže (Obr. 8.1).

Obrázky hotelů								
Id	Hotel ▲	V...	Statický soubor	Cestovní kancelář	Poslední aktualizace	Hash	Šířka	Výška
285...	Ananri Hotel & Spa [12868]	0		Tjaereborg (DE)	2017-12-06 15:46:48	8E221E0C377E9C147CE626948D...	700	442
285...	Aanari Hotel & Spa [12868]	0		JT Touristik	2017-12-06 15:46:48	19D049AF8CDAEB4A0622C1EB13...	700	442

Obr. 8.1 Náhled administrace uložených fotografií

8.1.1 Výpočet poměrného počtu hran

Jedná se o matematickou operaci založenou na kovoluci, DFT a IFT. Základní torzonošného algoritmu je vidět níže (Kod. 8.1).

Kod. 8.1 SharpnessCore.java

```

1 package cz.sa.dovolena.similarityclient.similarity;
2 ...
3 ...
4 import edu.emory.mathcs.jtransforms.fft.DoubleFFT_2D;
5 import javax.imageio.ImageIO;
6 ...
7 ...
8 ...
9 ...
10 /**
11 * Pracuje s konvolucním jadrem 3x3 typu horní propust, symetrickým, ne ideálním

```

```

12 * => propousti vysoké prostorové frekvence (nízké zahazuje)
13 *
14 * @author Dobroslav Pelc
15 */
16 public class SharpnessCounter {
17 ...
18 /**
19 * Určí poměrný koeficient ostrosti
20 *
21 * @return koeficient ostrosti
22 */
23 public double countSharpness() {
24     log.debug("Sharpness: " + filePath);
25     File modelImageFile = new File(filePath);
26     Preconditions.checkNotNull(modelImageFile.exists(), "[" + filePath + "] → obrázek musí existovat.");
27     Preconditions.checkArgument(modelImageFile.canRead(), "[" + filePath + "] → nad obrázkem musí být právo ke čtení.");
28     /* Konstante scalovány vzorový obrázek */
29     RenderedImage image = readBufferedImage(modelImageFile);
30     matrixXSize = image.getHeight();
31     matrixYSize = image.getWidth() * 2;
32     Preconditions.checkArgument(matrixXSize >= KERNEL_SIZE && matrixYSize / 2 >= KERNEL_SIZE, "[" + filePath + "] → obrázek musí být > ="
33     + " než jádro konvoluční matici tj. " + KERNEL_SIZE + "x" + KERNEL_SIZE + " px");
34
35     fFTTransformer = new DoubleFFT_2D(matrixXSize, matrixYSize / 2);
36     /* Obrázek preveden do matic, px = cross corel value */
37     double[][] fTMatrix = buildBrightnessMatrix(image);
38     /* Původní pole preberne novou hodnotu z fft transformera */
39     fFTTransformer.realForwardFull(fTMatrix);
40     /* Konvoluční jádro ve vlnovém spektru přes horní propust */
41     double[][] ftHighKernelMatrix = buildFTHighKernelMatrix();
42     /* Konvoluční jádro ve vlnovém spektru přes dolní propust */
43     double[][] ftLowKernelMatrix = buildFTLowKernelMatrix();
44     /* konvoluce ve vlnovém spektru s jádrem horní propust */
45     double[][] convolutionHighMatrix = buildConvolutionMatrix(fTMatrix, ftHighKernelMatrix);
46     /*
47     * Odfiltruje sumu ve vysokých frekvencích:
48     * konvoluce ve vlnovém spektru s jádrem horní propust a následně i s
49     * jádrem dolní propust
50     */
51     double[][] convolutionHighLowMatrix = buildConvolutionMatrix(convolutionHighMatrix, ftLowKernelMatrix);
52     /* zpětna (inverzní) FT */
53     /* konvoluce v původním souradnicovém systému */
54     fFTTransformer.complexInverse(convolutionHighMatrix, true);
55     fFTTransformer.complexInverse(convolutionHighLowMatrix, true);
56
57     sharpness = sumSharpnessPointOfBorder(convolutionHighMatrix);
58     overSharpness = sumOverSharpnessPointOfBorder(convolutionHighMatrix, convolutionHighLowMatrix);
59
60     return (double) sharpness / Math.max(OVER_SHARPNESS_ERROR_BOUND, overSharpness - OVER_SHARPNESS_ERROR_BOUND);
61 }
62 ...
63 ...
64 ...
65 }
66 }
```

Světlostní matice vznikne převodem fotografie (Obr. 1.1) do dvourozměrného pole (Kod. 8.2).

Kod. 8.2 BrightnessMatrix.java

```

1 /**
2  * Vytvoří matici, která má pro každý px obrázku vypočítat průmernou světlost
3  * a uloží do výstupního pole. Pokud obrázek není v poměru 4:3, prázdné
4  * místa se doplní nulami.
5  *
6  * @param image vyrenderovaný obrázek
7  * @return pole křízové korelace všech px obrázku.
8  */
9 private double[][] buildBrightnessMatrix(RenderedImage image) {
10
11     Preconditions.checkNotNull(image, "Obrazek, nad kterým chcete vypočítat průmernou světlost, nesmí být null.");
12     final int imageWidth = image.getWidth();
13     Preconditions.checkArgument(imageWidth <= matrixYSize / 2, "Obrazek, nad kterým chcete vypočítat průmernou světlost, musí být scalovány na 320x240px.");
14     final int imageHeight = image.getHeight();
15     Preconditions.checkArgument(imageHeight <= matrixXSize, "Obrazek, nad kterým chcete vypočítat průmernou světlost, musí být scalován na 320x240px.");
16
17     final double[][] brightnessMatrix = new double[matrixXSize][matrixYSize];
18     final RandomIter iterator = RandomIterFactory.create(image, null);
19
20     for (int x = 0; x < imageHeight; x++) {
21         for (int y = 0; y < imageWidth; y++) {
22             final double[] pixel = new double[RGB];
23             /* Pro vchodnéjsí práci s maticí v FFT jsme otocili význam indexu, z obrázku ale musíme násobit pořad stejné */
24             iterator.getPixel(y, x, pixel);
25
26             /* Vypočet jasu průměrného jasu pixelu */
27             double averageBrightness = (pixel[0] + pixel[1] + pixel[2]) / 3;
28             brightnessMatrix[x][y] = averageBrightness;
29         }
30     }
31     return brightnessMatrix;
32 }
```

Matrice ostrosti je matice hran, která se získá převodem světlostní matice do vlnového spektra pomocí DFT a následnou konvolucí ve vlnovém spektru (Kod. 8.3). Grafická prezentace této operace je na obrázku níže (Obr. 2.1). K docílení požadovaného stavu

je nutné zvolit vhodný filtr (Tab. 8.1, (Kod. 8.3)). Samotné sestavení matice ostrosti ve vlnovém spektru znázorňuje přiložený kód níže (Kod. 8.4).

Kod. 8.3 FTHighKernelMatrix.java

```

1  /* POZN: součet všech zaporných a kladných koeficientu musí být = 0. */
2  /* Rozměr konvolučního jádra – zpravidla čtvercové. */
3  public static final int KERNEL_SIZE = 3;
4  /* Negativní koeficient konvolučního jádra pro horní propust */
5  private static final double HIGH_KERNEL_COEF_NEGATIV = -0.0625; // -(1/16)
6  /* Kladný koeficient konvolučního jádra pro horní propust */
7  private static final double HIGH_KERNEL_COEF_POSITIVE = 0.5;
8
9
10 private double[][] buildFTHighKernelMatrix() {
11     /* Druhá matici z konvolučního jádra */
12     double[][] kernelMatrix = new double[matrixXSize][matrixYSize];
13     /*
14      * do prvních ×33 polí dame konvoluční jádro:
15      *
16      *   -1/16   -1/16   -1/16
17      *   -1/16    1/2    -1/16
18      *   -1/16   -1/16   -1/16
19      */
20     kernelMatrix[0][0] = HIGH_KERNEL_COEF_POSITIVE;
21     kernelMatrix[0][1] = HIGH_KERNEL_COEF_NEGATIV;
22     kernelMatrix[1][0] = HIGH_KERNEL_COEF_NEGATIV;
23     kernelMatrix[1][1] = HIGH_KERNEL_COEF_NEGATIV;
24     kernelMatrix[matrixXSize - 1][0] = HIGH_KERNEL_COEF_NEGATIV;
25     kernelMatrix[matrixXSize - 1][1] = HIGH_KERNEL_COEF_NEGATIV;
26     kernelMatrix[0][(matrixYSize / 2) - 1] = HIGH_KERNEL_COEF_NEGATIV;
27     kernelMatrix[1][(matrixYSize / 2) - 1] = HIGH_KERNEL_COEF_NEGATIV;
28     kernelMatrix[matrixXSize - 1][(matrixYSize / 2) - 1] = HIGH_KERNEL_COEF_NEGATIV;
29
30     fFTransformer.realForwardFull(kernelMatrix);
31     return kernelMatrix;
32 }
```

Tab. 8.1 Jádro typu horní propust

-1/16	-1/16	-1/16
-1/16	1/2	-1/16
-1/16	-1/16	-1/16

Kod. 8.4 ConvolutionMatrix.java

```

1 /**
2  * Konvoluce ve vlnovém spektru.
3  *
4  * @param a matice FT z obrazku
5  * @param b matice FT konvolučního jádra
6  * @return a .* b' (vynásobí prvek po prvku a a b)
7  */
8 protected double[][] buildConvolutionMatrix(double[][] a, double[][] b) {
9     double[][] result = new double[matrixXSize][matrixYSize];
10    for (int x = 0; x < matrixXSize; x++) {
11        for (int y = 0; y < matrixYSize / 2; y++) {
12            final double realPartA = a[x][2 * y];
13            final double realPartB = b[x][2 * y];
14            final double imagPartA = a[x][2 * y + 1];
15            final double imagPartB = b[x][2 * y + 1];
16            /* reálna cast výsledné matice */
17            result[x][2 * y] = realPartA * realPartB - imagPartA * imagPartB;
18            /* imaginární cast výsledné matice */
19            result[x][2 * y + 1] = realPartA * imagPartB + imagPartA * realPartB;
20        }
21    }
22    return result;
23 }
```

Matice přeostření je matice hran za prahovou hodnotou ostrosti (přeostřených hran). Vzniká z konvolucí (Kod. 8.4) matice ostrosti s jádrem typu dolní propust (Tab. 8.2, Kod. 8.5) bylo sestaveno experimentálně v mnoha iteracích a jeho funkčnost ověřena testováním na lidech, stejně jako aplikovaná prahová hodnota.

Kod. 8.5 FTLowKernelMatrix.java

```

1  /* Koeficient okolních bodu konvolučního jádra pro dolní propust */
2  private static final double LOW_KERNEL_NEIGHBOUR_COEF = 0.125; // 1/8
3  /* Středový koeficient konvolučního jádra pro dolní propust */
4  private static final double LOW_KERNEL_MIDDLE_COEF = 0.5;
5
6  private double[][] buildFTLowKernelMatrix() {
7      /*
8       * do prvních 3x3 polí dame konvoluční jádro
9       *
10      *   1/8    1/8    1/8
11      *   1/8    1/2    1/8
12      *   1/8    1/8    1/8
13      */
14  double[][] kernelMatrix = new double[matrixXSize][matrixYSize];
15
16  kernelMatrix[0][0] = LOW_KERNEL_MIDDLE_COEF;
17  kernelMatrix[0][1] = LOW_KERNEL_NEIGHBOUR_COEF;
18  kernelMatrix[1][0] = LOW_KERNEL_NEIGHBOUR_COEF;
19  kernelMatrix[matrixXSize - 1][0] = LOW_KERNEL_NEIGHBOUR_COEF;
20  kernelMatrix[0][(matrixYSize / 2) - 1] = LOW_KERNEL_NEIGHBOUR_COEF;
21
22  fFTransformer.realForwardFull(kernelMatrix);
23  return kernelMatrix;
24 }

```

Tab. 8.2 Jádro typu dolní propust'

1/8	1/8	1/8
1/8	1/2	1/8
1/8	1/8	1/8

Poměrný počet hran je podíl sumy hran a sumy přeostřených hran. Suma hran je dána součtem jednotlivých prvků matice ostrosti po převodu zpět z vlnového spektra pomocí IFT (Kod. 8.6). Suma přeostřených hran vzniká podobně (Kod. 8.7). Jediný rozdíl je zohlednění prahového hodnoty ostrosti (respektive přeostřenosti).

Kod. 8.6 sumSharpness.java

```

1 /**
2  * Sesteme body na hranach, které jsou dostatečně ostre. Za míru ostrosti
3  * povazujeme ostrost obrazu, která je závislá na jeho velikosti.
4  *
5  * @param matrix matici hran
6  * @return "míra ostrosti"
7  */
8 private int sumSharpnessPointOfBorder(double[][] matrix) {
9     return sumOverSharpnessPointOfBorder(matrix, new double[matrixXSize][matrixYSize]);
10 }

```

Kod. 8.7 sumOverSharpness.java

```

1 /* Prah preostřených hran */
2 private static final int BORDER_SHARPNESS_LIMIT = 20;
3
4 /**
5  * Četeme body na hranách, které jsou čestotatně řepeosten.
6  *
7  * @param matrixA matici hran
8  * @param matrixB matici hran bez šumu (áprojet idoln ípropust)
9  * @return imra řepeostenosti
10 */
11 private int sumOverSharpnessPointOfBorder(double[][] matrixA, double[][] matrixB) {
12     int sum = 0;
13     for (int x = 0; x < matrixXSize; x++) {
14         for (int y = 0; y < matrixYSize / 2; y++) {
15             if (matrixA[x][y * 2] - matrixB[x][y * 2] > BORDER_SHARPNESS_LIMIT) {
16                 sum++;
17             }
18         }
19     }
20     /* imra řepeostenosti */
21     return sum;
22 }

```

Příklad přeostřenosti fotografie prezentují obrázky níže. Z původní fotografie (Obr. 1.1) je po oddělení přeostřených hran (Obr. 3.1), ode všech hran (Obr. 2.1) stanoven jejich poměrný koeficient.

8.2 Fronta nezpracovaných obrázků

Obrázky, které čekají na výpočet KoP, lze vyčíst z DB pomocí SQL dotaz (Kod. 8.8). Priorita nezpracovaných obrázků je dána počtem zhlédnutí webové stránky hotelu. Dávka 10.000 požadavků na výpočet KoP, která je jednorázově načtena z DB, vystačí přibližně na 10 sekund (v závislosti na počtu aktivních slave). Počty shlédnutí jednotlivých hotelů se propisují do DB v reálném čase. Každá nově načtená dávka tudíž zohlední prioritu KoP.

Kod. 8.8 queue.sql

```

1  SELECT
2      pattern.photo AS pattern,
3      compare.photo AS compare,
4      h.visited AS priority
5
6  FROM hotel AS h
7  JOIN hotel_photo AS pattern ON pattern.hotel = hotel.id
8  JOIN hotel_photo AS compare ON compare.hotel = hotel.id AND pattern.photo <> compare.photo
9
10 /* Hledáme pouze fotky, u kterých není sponcitá podbnost; záleží na poradí! */
11 LEFT JOIN similarity AS s ON (s.pattern = pattern.photo AND s.compare = compare.photo)
12     OR (s.pattern = compare.photo AND s.compare = pattern.photo)
13
14 WHERE s IS NULL
15 ORDER BY h.visited DESC NULL LAST
16 LIMIT 10000;

```

8.3 Servlet pro stažení obrázků

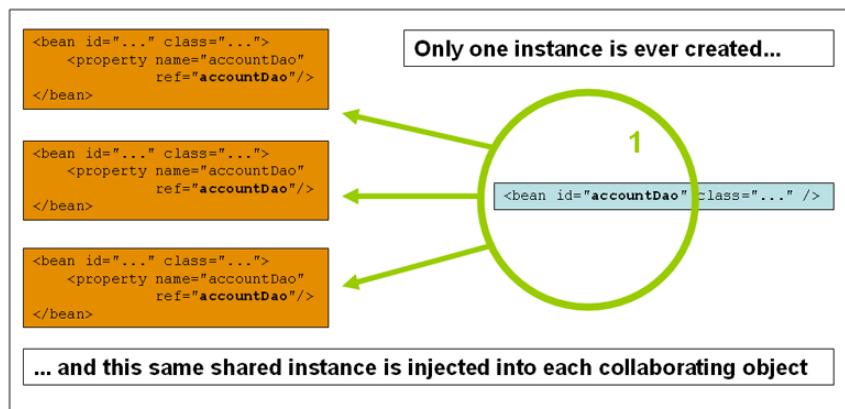
Servlet je součást serverové strany. Je realizován pomocí spring bean [35] typu singleton [2] (Obr. 8.2). Na základě vstupní URL adresy [33] (GET dotazu [34]) určí parametr md5. Pomocí něj vyčte z DB relativní cestu na datovém, síťovém úložišti, kde se fotografie nachází (Kod. 8.9).

Kod. 8.9 servlet.sql

```

1  SELECT relative_path
2  FROM photo
3  WHERE md5 = $arg;

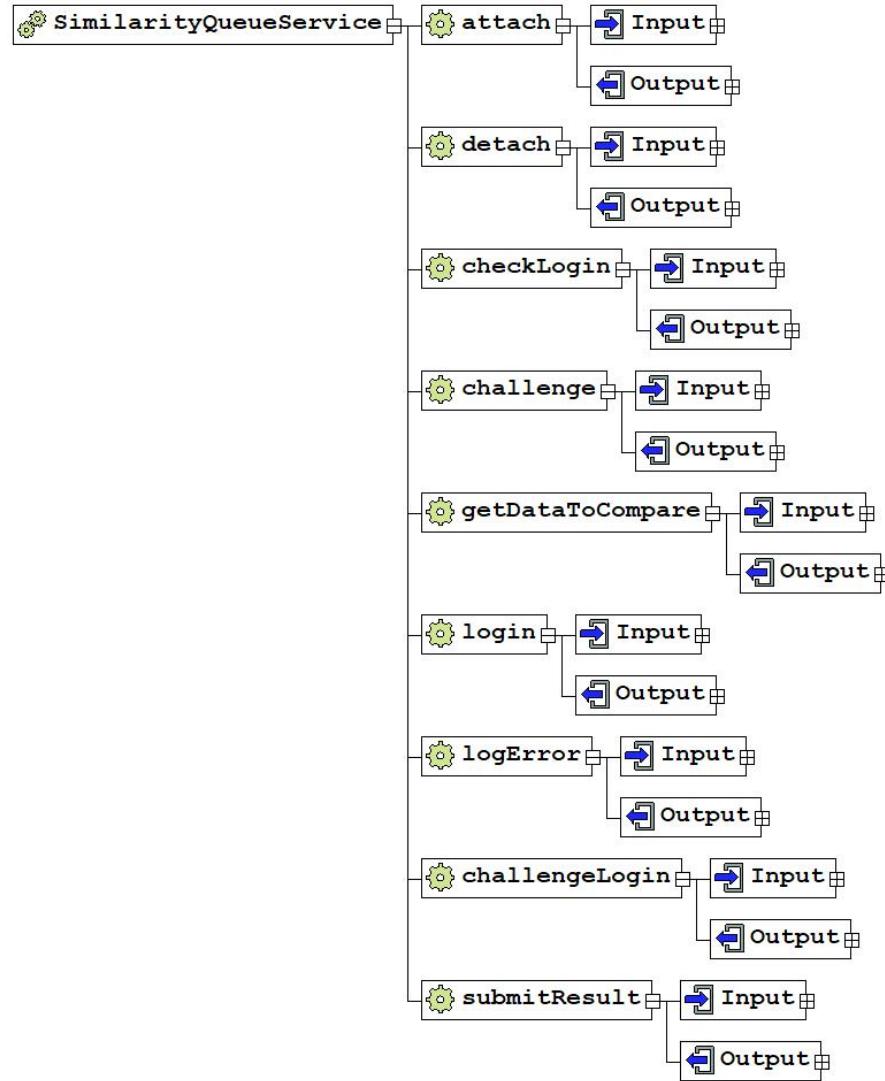
```



Obr. 8.2 Ukázka použití singleton [36]

8.4 SOAP API

Kompletní komunikační rozhraní ve formátu WSDL [28] je k dispozici na přiloženém CD. Vybranou část WSDL znázorňuje obrázek níže (Obr. 8.3).



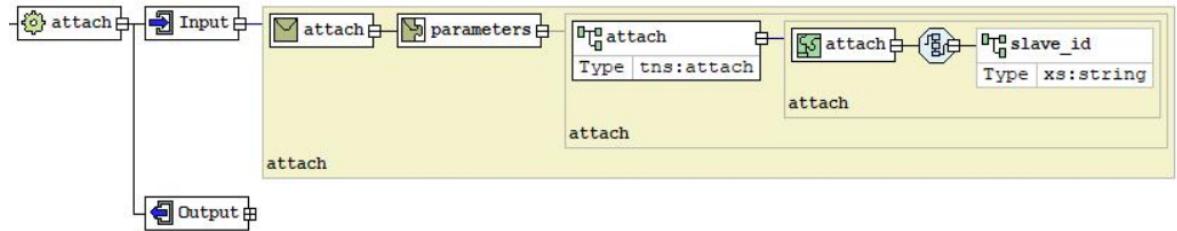
Obr. 8.3 Vizualizace komunikačních metod distribuované služby

8.4.1 Autentizace

Aby bylo možné konzumovat vystavené webové služby, je nejprve nutné provést autentizaci. K těmto účelům je použita technika CRAM [30] (někdy také jako challenge-response).

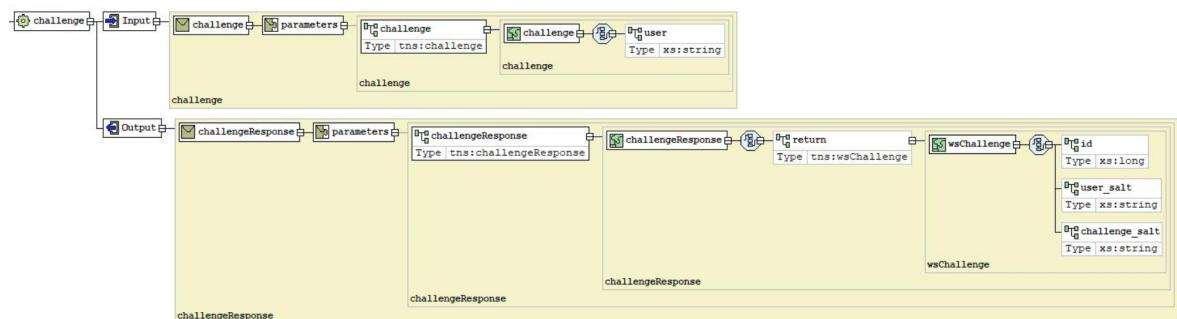
attach je metoda, která slouží k navázání vyhrazeného komunikačního kanálu (relace nebo také session [31]) mezi serverem a strojem pro odbavování distribuovaných požadavků (dále jen slave) (Obr. 8.4). Je nezabezpečená metoda (nevyžaduje autentizaci).

Slouží pouze k zaregistrování slave_id (jedinečný identifikátor slave) na straně serveru.



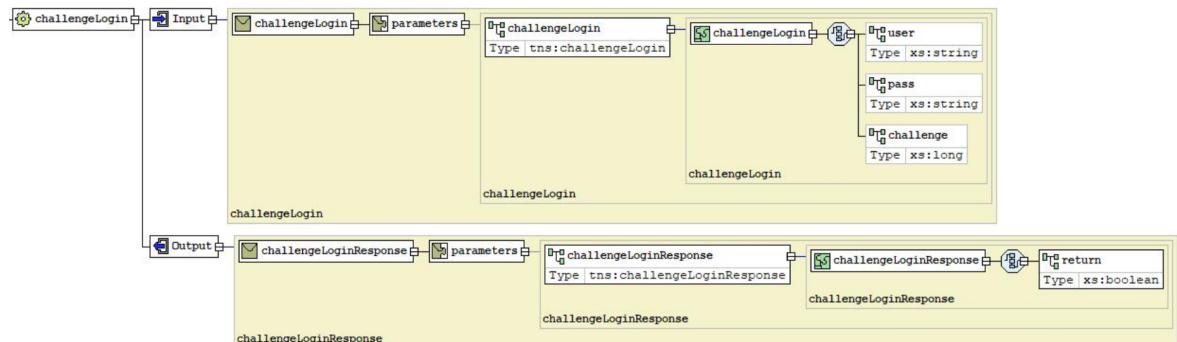
Obr. 8.4 Detail metody attach

challenge je metoda, kterou slave žádá server o vytvoření zabezpečeného komunikačního kanálu. Server na tento požadavek reaguje pouze v případě, že se slave prokáže platným slave_id. V odpovědi server vrací vygenerovanou sůl (pseudonáhodné hexadecimální číslo [32]) potřebnou pro další komunikaci (Obr. 8.5).



Obr. 8.5 Detail metody challenge

challengeLogin je metoda pro autentizaci slave. Server i slave provedou nezávisle na sobě výpočet hashe (s přednastaveným hashovacím algoritmem) za použití soli. Slave odešle data (Obr. 8.6) a pokud jsou ve shodě se serverovými, je relace tohoto slave nadále autentizována.

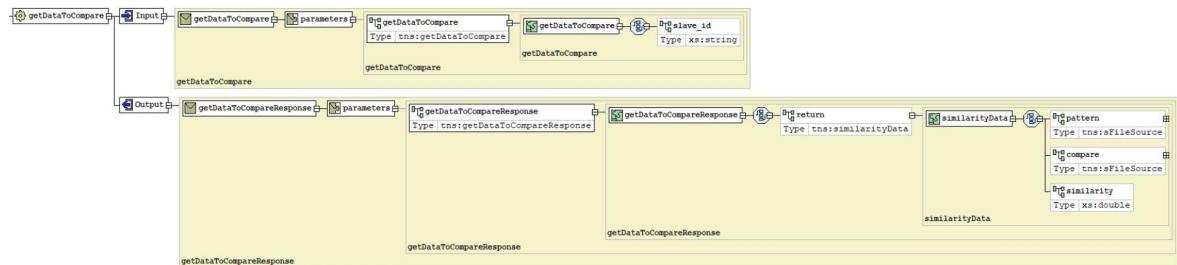


Obr. 8.6 Detail metody challengeLogin

8.4.2 Distribuce požadavků na výpočet KoP

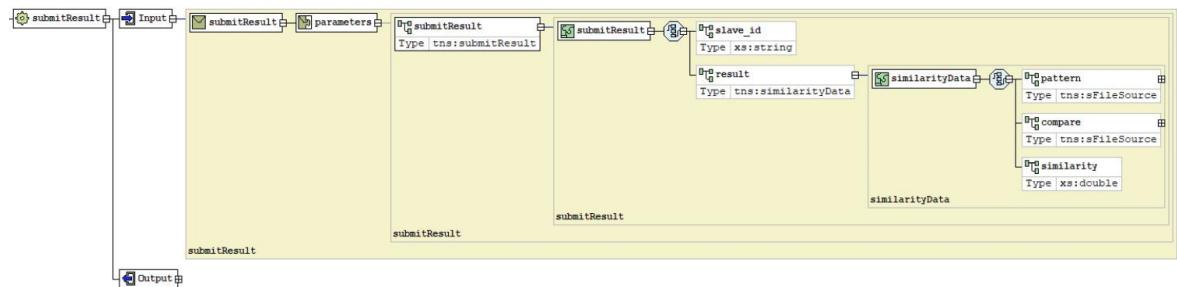
Autentizovaný slave může zažádat o metadata k výpočtu KoP. Pokud jsou ve frontě na výpočet KoP zařazeny nějaké požadavky, jsou vráceny. Jakmile je slave vyhodnotí, odesílá výsledek zpět na server.

getDataToCompare je metoda určená pro distribuci výpočtu (metadat k výpočtu) KoP (Obr. 8.7).



Obr. 8.7 Detail metody *getDataToCompare*

submitResult je metoda určená pro odeslání výsledku distribuovaného výpočtu KoP zpět na server (Obr. 8.8).



Obr. 8.8 Detail metody *submitResult*

9 Klientská (výkonná) strana distribuované služby

Kod. 9.1 ImageComparatorWithConstantScale.java

```

1 package cz.sa.dovolena.similarityclient.similarity;
2 import com.google.common.base.Preconditions;
3 import edu.emory.mathcs.jtransforms.fft.DoubleFFT_2D;
4 import java.awt.image.BufferedImage;
5 import java.awt.image.RenderedImage;
6 import java.awt.image.renderable.ParameterBlock;
7 import java.io.File;
8 import java.io.IOException;
9
10 import javax.imageio.ImageIO;
11 import javax.media.jai.InterpolationNearest;
12 import javax.media.jai.JAI;
13 import javax.media.jai.RenderedOp;
14 import javax.media.jai.iterator.RandomIter;
15 import javax.media.jai.iterator.RandomIteratorFactory;
16 import org.apache.log4j.Logger;
17
18 /**
19 * éScaleVzorovo a čírefennch áobrzk na [320 x ?] nebo [? x 240] + čídoplnn
20 * évzorovo i číreferenchno nulami na [320 x 240]. áDle provede řízkovou
21 * korelace ve évnovn spektru. <B>Operace je ikomutativn.</B>
22 *
23 * @author Dobroslav Pelc, 05.12.2014 9:42
24 */
25
26 public class ImageComparatorWithConstantScale {
27
28     /* žáPořadová šířka áobrzk */
29     protected static final int BASE_SIZE_WIDTH = 320;
30     /* žáPořadová ýška áobrzk */
31     protected static final int BASE_SIZE_HEIGHT = 240;
32
33     /* čPoet barev pro ýzvolen ýgrafick žreim */
34     private static final int RGB = 3;
35
36     protected int matrixXSize = BASE_SIZE_HEIGHT;
37     protected int matrixYSize = 2 * BASE_SIZE_WIDTH;
38
39     protected DoubleFFT_2D fFTTransformer;
40     protected double[][] normalizedFourierMatrix;
41
42     protected final RenderedImage scaledModelImage;
43
44     private final Logger log = Logger.getLogger(getClass());
45
46     /**
47      * @param originalFilePath cesta k évzorovmu áobrzk
48     */
49     public ImageComparatorWithConstantScale(String originalFilePath) {
50         Preconditions.checkNotNull(originalFilePath, "Cesta ke ézdrojovmu souboru ímus ýbt éavyplnn.");
51         File modelImageFile = new File(originalFilePath);
52         Preconditions.checkArgument(modelImageFile.exists(), "[" + originalFilePath + "] —> ývzorov áobrzek ímus existovat.");
53         Preconditions.checkArgument(modelImageFile.canRead(), "[" + originalFilePath + "] —> nad ývzorovm áobrzkem ímus ýbt áprvo ke čiten.");
54         /* ékonstantn ýscalovan ývzorov áobrzek */
55         scaledModelImage = doConstantScale(modelImageFile);
56     }
57
58     public void bindModel() {
59         fFTTransformer = new DoubleFFT_2D(matrixXSize, matrixYSize / 2);
60         double[][] matrixImage = buildNormalizedFTMatrix(scaledModelImage);
61         /* ýáVsledn matic */
62         normalizedFourierMatrix = matrixImage;
63     }
64
65     protected RenderedImage doConstantScale(File modelImageFile) {
66         RenderedImage scaledModelImage = scale(
67             readBufferedImage(modelImageFile),
68             BASE_SIZE_WIDTH,
69             BASE_SIZE_HEIGHT
70         );
71
72         return scaledModelImage;
73     }
74
75     protected BufferedImage readBufferedImage(File modelImageFile) {
76         try {
77             return ImageIO.read(modelImageFile);
78         } catch (IOException ex) {
79             log.fatal("Nepodaílo se činast áobrzek " + modelImageFile.getAbsolutePath() + " ", ex);
80         }
81         return null;
82     }
83
84     protected double[][] buildNormalizedFTMatrix(RenderedImage scaledModelImage) {
85         /* áobrzek rýpeveden do matic, px = cross corel value */
86         double[][] matrixImage = buildBrightnessMatrix(scaledModelImage);
87         /* áobrzek je žij rýpeveden furierovu transformac */
88         /* ÚPvodn pole řepbere novou hodnotu */
89         fFTTransformer.realForwardFull(matrixImage);
90         /* áíNormovn Fourierovi transformace áobrzk */
91         for (int x = 0; x < matrixXSize; x++) {
92             for (int y = 0; y < matrixYSize / 2; y++) {
93                 /* bereme ésd y => áéreln číslo */
94                 final int realPartOfMatrixFieldY = 2 * y;
95                 final double realPart = matrixImage[x][realPartOfMatrixFieldY];
96
97                 /* bereme élich y => áíimaginrn číslo */
98                 final int imagPartOfMatrixFieldY = 2 * y + 1;
99                 final double imagPart = matrixImage[x][imagPartOfMatrixFieldY];
100
101                /* iabsolutn hodnota íkomplexnho čísla */
102                double absoluteMatrixFieldValue = Math.sqrt(imagPart * imagPart + realPart * realPart);
103
104                /*
105                 * Pokud ývde nula, tak obecne doslo k tomu, ze pro nejakou
106                 * mochnu dvou 2^k je v nejakym radku nebo sloupce kazdej
107                 * 2^k-ý pixel stejnej
108                 */
109                if (absoluteMatrixFieldValue != 0) {

```

```

110         Preconditions.checkNotNull(absoluteMatrixFieldValue != 0, "Nulou nelze èdit.");
111         matrixImage[x][realPartOfMatrixFieldY] /= absoluteMatrixFieldValue;
112         matrixImage[x][imagPartOfMatrixFieldY] /= absoluteMatrixFieldValue;
113     }
114 }
115 return matrixImage;
116 }
117 }
118 /**
119 * áíVrt ývrenderovan áobrzek v ýnoch érozmrech. Kontroluje èívt stranu,
120 * podle ékter číur épomr. NEDEFORMUJE épomr stran => ýVsledn áobrzek je
121 * šejet èpoteba doplnit nulama, žprote inemus iòpovdat řepes ýzadan
122 * éurozmr m v obouch áosch.scaledImage@param image áioriginln áobrzek
123 *
124 * @param scaleToWidth žýpoadovan šíèka, do ékter se ám áobrzek scalovat
125 * @param scaleToHeight žápoadovan ývka, do ékter se ám áobrzek scalovat
126 *
127 * @return áobrzek v ýnoch érozmrech
128 */
129 protected RenderedImage scale(RenderedImage image, int scaleToWidth, int scaleToHeight) {
130     RenderedOp rescale = null;
131     final int originalWidth = image.getWidth();
132     final int originalHeight = image.getHeight();
133
134     float widthRatio = (float) scaleToWidth / (float) originalWidth;
135     float heightRatio = (float) scaleToHeight / (float) originalHeight;
136
137     float ratio = Math.min(widthRatio, heightRatio);
138
139     // Scales the original image
140     ParameterBlock pb = new ParameterBlock();
141     pb.addSource(image);
142     pb.add(ratio);
143     pb.add(0.0F);
144     pb.add(0.0F);
145     pb.add(new InterpolationNearest());
146
147     // Creates a new, scaled image and uses it on the DisplayJAI component
148     rescale = JAI.create("scale", pb);
149
150     return rescale;
151 }
152 /**
153 * říVytvo matici, ákter ám pro žýkad px áobrzu číavypot ùéprmrnou èsvtlost
154 * a žiulo do ývstupnho pole. Pokud áobrzek inen v épomru ×43, áeprzd
155 * ímsta se ídopl nulami.
156 *
157 * @param image ývrenderovan áobrzek
158 * @return pole řížekv korelace švech px áobrzu.
159 */
160 private double[][] buildBrightnessMatrix(RenderedImage image) {
161
162     Preconditions.checkNotNull(image, "áobrzek, nad ýkterch chcete číavypotat ùéprmrnou èsvtlost, ínesm ýbt null.");
163     final int imageWidth = image.getWidth();
164     Preconditions.checkArgument(imageWidth <= matrixYSize / 2, "áobrzek, nad ýkterch chcete číavypotat ùéprmrnou èsvtlost, ímus ýbt ýscalovan
165     na <32024px.");
166     final int imageHeight = image.getHeight();
167     Preconditions.checkArgument(imageHeight <= matrixXSize, "áobrzek, nad ýkterch chcete číavypotat ùéprmrnou èsvtlost, ímus ýbt ýscalovan na
168     <32024px.");
169
170     final double[][] brightnessMatrix = new double[matrixXSize][matrixYSize];
171     final RandomIter iterator = RandomIterFactory.create(image, null);
172
173     for (int x = 0; x < imageHeight; x++) {
174         for (int y = 0; y < imageWidth; y++) {
175             final double[] pixel = new double[RGB];
176             /* Pro èívhodnj áprci s imatic v FFT jsme čotoili ývnam ūindex, z áobrzu ale ímusm činataf řapod èstejn */
177             iterator.getPixel(y, x, pixel);
178
179             /* ýVpoet jasu ùéprmrnho jasu pixelu */
180             double averageBrightness = (pixel[0] + pixel[1] + pixel[2]) / 3;
181             brightnessMatrix[x][y] = averageBrightness;
182         }
183     }
184
185     return brightnessMatrix;
186 }
187 /**
188 * áPorovn ývstupn áobrzek se vzorem. Je pravidlem, že ýsledek áiporovnn
189 * évzorovho áobrzu s čireferennm je ýstejn i naopak. Proto je èvodn
190 * áeprovdt áiporovn žýkad s žýkadm, kdy ážiNEzle na řípad.
191 *
192 * @param path íabsolutn cesta k édruhmu áobrzu
193 * @return koeficient podobnosti áobrzu účvi vzoru
194 */
195 public double compare(String path) {
196     Preconditions.checkNotNull(path, "Cesta ke édrojovmu souboru ímus ýbt éavyplnn.");
197     File reference = new File(path);
198     Preconditions.checkNotNull(reference.exists(), "ýVzorov áobrzek ímus existovat.");
199     Preconditions.checkNotNull(reference.canRead(), "Nad ývzorovm áobrzkem ímus ýbt áprvo ke čiten.");
200
201     try {
202         /* èKonstantn ýscalovan čireferenn áobrzek */
203         final RenderedImage comparedImage = doConstantScale(reference);
204         /* ýVsledn maticé édruhho áobrzu */
205         double[][] matrixImage = buildNormalizedFTMatrix(comparedImage);
206
207         /* řížakov korelace ve ývnovm spektru*/
208         double[][] crossCorelMatirx = crossCorrelation(normalizedFourierMatrix, matrixImage);
209
210         /* èázptn (íinverzn FT */
211         /* řížakov korelace v ývpodnm řésouadnicovm éysystmu */
212         FFTTransformer.complexInverse(crossCorelMatirx, true);
213
214         return maxCorrelationCoefficient(crossCorelMatirx);
215     } catch (Throwable e) {
216         log.fatal("řPi ýcpotu podobnosti " + path + " šdolo k èchyb.", e);
217         return -1D;
218     }
219 }
220 }

```

```
222 public double maxCorrelationCoefficient(double[][] matrix) {
223     double max = 0;
224     for (int x = 0; x < matrixXSize; x++) {
225         for (int y = 0; y < matrixYSize / 2; y++) {
226             max = Math.max(
227                 Math.abs(matrix[x][y * 2]),
228                 max);
229         }
230     }
231     return max;
232 }
233 /**
234 * řížáKov korelace ve évlnovm spektru.
235 *
236 * @param a ánormovan matice FT vzor
237 * @param b ánormovan matice FT obrazu
238 * @return a .* b' (áivynsobm prvek po prvk a a b, řčpiem b je ēkomplexn
239 * žázrduen)
240 */
241 protected double[][] crossCorrelation(double[][] a, double[][] b) {
242     double[][] result = new double[matrixXSize][matrixYSize];
243     for (int x = 0; x < matrixXSize; x++) {
244         for (int y = 0; y < matrixYSize / 2; y++) {
245             final double realPartA = a[x][2 * y];
246             final double realPartB = b[x][2 * y];
247             final double imagPartA = a[x][2 * y + 1];
248             final double imagPartB = b[x][2 * y + 1];
249             /* áareln část ýévsledn matice */
250             result[x][2 * y] = realPartA * realPartB + imagPartA * imagPartB;
251             /* áímaginrn část ýévsledn matice */
252             result[x][2 * y + 1] = realPartA * imagPartB - imagPartA * realPartB;
253         }
254     }
255     return result;
256 }
257 }
258 }
259 }
260 }
```

ZÁVĚR

Text závěru

SEZNAM POUŽITÉ LITERATURY

- [1] ECKEL, Bruce. *Thinking in Java*. 4th ed. Upper Saddle River, NJ: Prentice Hall, c2006. ISBN 01-318-7248-6.
- [2] WALLS, Craig. *Spring in action*. Fourth edition. Texas: Manning, 2013. ISBN 978-161-7291-203.
- [3] HUNT, Charlie a Binu JOHN. *Java performance*. Upper Saddle River, NJ: Addison-Wesley, c2012. Java series. ISBN 01-371-4252-8.
- [4] PERŮTKA, Karel. *MATLAB - Základy pro studenty automatizace a informačních technologií*. Zlín: Univerzita Tomáše Bati ve Zlíně, 2005. ISBN 80-731-8355-2.
- [5] DOBEŠ, Michal. *Zpracování obrazu a algoritmy v C#*. Praha: BEN - technická literatura, 2008. ISBN 978-80-7300-233-6.
- [6] MD5. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-08]. Dostupné z: <https://en.wikipedia.org/wiki/MD5>.
- [7] Computer vision [online]. [cit. 2018-05-08]. Dostupný z: http://midas.uamt.feec.vutbr.cz/ZVS/Exercise02/content_cz.php.
- [8] Pixel. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-08]. Dostupné z: <https://cs.wikipedia.org/wiki/Pixel>.
- [9] RGB. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-08]. Dostupné z: <https://cs.wikipedia.org/wiki/RGB>.
- [10] IEEE Transactions on Computers [online]. 1972, **C-21**(2), 179–186 [cit. 2018-05-08]. DOI: 10.1109/TC.1972.5008923. ISSN 0018-9340. Dostupné z: <http://ieeexplore.ieee.org/document/5008923/>.
- [11] Korelace. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-09]. Dostupné z: <https://cs.wikipedia.org/wiki/Korelace>.
- [12] Cross-correlation. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-09]. Dostupné z: <https://en.wikipedia.org/wiki/Cross-correlation>.

- [13] *FFTW* [online]. [cit. 2018-05-08]. Dostupný z: <http://www.fftw.org>.
- [14] The Design and Implementation of FFTW3. In: *Proceedings of the IEEE* [online]. 2005, **93**(2), s. 216–231 [cit. 2018-05-08]. DOI: 10.1109/JPROC.2004.840301. ISSN 0018-9219. Dostupné z: <http://ieeexplore.ieee.org/document/1386650>.
- [15] *IEEE Transactions on Image Processing* [online]. **5**(8), 1266-1271 [cit. 2018-05-08]. DOI: 10.1109/83.506761. ISSN 10577149. Dostupné z: <http://ieeexplore.ieee.org/document/506761/>.
- [16] *Shared Scientific Toolbox in Java* [online]. [cit. 2018-05-08]. Dostupné z: <http://freshmeat.sourceforge.net/projects/shared>.
- [17] *CMake* [online]. [cit. 2018-05-08]. Dostupný z: <https://cmake.org>.
- [18] Wrapper. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-09]. Dostupné z: <https://en.wikipedia.org/wiki/Wrapper>.
- [19] In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: <https://cs.wikipedia.org/wiki/Konvolute>.
- [20] Detekce hran. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: https://cs.wikipedia.org/wiki/Detekce_hran.
- [21] SOAP. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: <https://en.wikipedia.org/wiki/SOAP>.
- [22] Centrální procesorová jednotka. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: https://cs.wikipedia.org/wiki/Centr%C3%A1ln%C3%AD_procesorov%C3%A1_jednotka.
- [23] Graphic processing unit. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: https://en.wikipedia.org/wiki/Graphics_processing_unit.
- [24] Java (programovací jazyk). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-10]. Dostupné z: [https://cs.wikipedia.org/wiki/Java_\(programovac%C3%AD_jazyk\)](https://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk)).

- [25] *CUDA Toolkit Documentation* [online]. March 5, 2018 [cit. 2018-05-16]. Dostupné z: <https://docs.nvidia.com/cuda/>.
- [26] Java Platform, Standard Edition. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-16]. Dostupné z: https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition.
- [27] *Spring* [online]. 2018 [cit. 2018-05-16]. Dostupné z: <https://spring.io/>.
- [28] Web Services Description Language. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-16]. Dostupné z: https://en.wikipedia.org/wiki/Web_Services_Description_Language.
- [29] *POSTGRESQL: THE WORLD'S MOST ADVANCED OPEN SOURCE RELATIONAL DATABASE* [online]. The PostgreSQL Global Development Group, 2018 [cit. 2018-05-16]. Dostupné z: <https://www.postgresql.org/>.
- [30] Challenge-response. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-16]. Dostupné z: <https://cs.wikipedia.org/wiki/Challenge-response>.
- [31] Session. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-16]. Dostupné z: <https://cs.wikipedia.org/wiki/Session>.
- [32] Sůl (kryptografie). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-16]. Dostupné z: [https://cs.wikipedia.org/wiki/S%C5%AFl_\(kryptografie\)](https://cs.wikipedia.org/wiki/S%C5%AFl_(kryptografie)).
- [33] URL. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-17]. Dostupné z: <https://en.wikipedia.org/wiki/URL>.
- [34] GET. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-17]. Dostupné z: <https://cs.wikipedia.org/wiki/GET>.
- [35] JavaBeans. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-17]. Dostupné z: <https://en.wikipedia.org/wiki/JavaBeans>.
- [36] Bean scopes [online]. [cit. 2018-05-17]. Dostupné z: <https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch04s04.html>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
CD	Compact Disc
CPU	Central Processing Unit
CRUM	Challenge-Response Authentication Mechanism
CUDA	Compute Unified Device Architecture
DFT	Discrete Fourier Transform (Diskrétní Fourierova transformace)
FFTW	Fastest Fourier Transform in the West
GPU	Graphic Processing Unit
HW	Hardware
IFT	Inverse Fourier Transform (Zpětná Fourierova transformace)
JavaSE	Java Platform, Standard Edition
KoP	Koeficient podobnosti dvou fotografií
KoZ	Koeficient zastupitelnosti vzájemně si podobných fotografií
MD5	Message-Digest algorithm
PC	Personal Computer
px	picture element (obrazový prvek)
SOAP	Simple Object Access Protocol
SW	Software
URL	Uniform Resource Locator
WSDL	Web Services Description Language

SEZNAM OBRÁZKŮ

2.1	Statický scaling obrázků (komutativní)	16
2.2	Dynamický scaling obrázků (diskomutativní)	16
2.3	Příklad proložení dvou fotografií s použitím klasické korelace	18
2.4	Příklad proložení dvou fotografií s použitím křížové korelace	18
3.1	Příklad Diskrétní 2D konvoluce [19]	20
6.1	Vizualizace procesu výpočtu koeficientů na CPU	32
6.2	Vizualizace procesu výpočtu koeficientů na CPU s paralelizací na GPU	33
7.1	L1 pohled aktuálního stavu	36
7.2	L1 pohled cílového stavu	36
7.3	Funkční požadavky	37
7.4	Nefunkční požadavky	37
7.5	Aktéři	38
7.6	Případy užití	38
7.7	Model tříd	39
7.8	Model služeb	39
7.9	Sekvenční diagram	40
8.1	Náhled administrace uložených fotografií	41
8.2	Ukázka použití singleton [36]	45
8.3	Vizualizace komunikačních metod distribuované služby	46
8.4	Detail metody attach	47
8.5	Detail metody challenge	47
8.6	Detail metody challengeLogin	47
8.7	Detail metody getDataToCompare	48
8.8	Detail metody submitResult	48
1.1	Fotografie před úpravou	61
2.1	Vizualizace matice hran	62
3.1	Vizualizace matice přeostřených hran	63

SEZNAM TABULEK

3.1	Jádro pro detekci hran (horizontálně a vertikálně)	21
3.2	Jádro pro detekci hran (horizontálně, vertikálně a šíkmé hrany)	21
5.1	Tabulka výsledků jednotlivých variant krátkodobého testu	26
5.2	Tabulka výsledků jednotlivých variant krátkodobého testu	26
8.1	Jádro typu horní propust̄	43
8.2	Jádro typu dolní propust̄	44

SEZNAM ZDROJOVÉHO KÓDU

6.1	banchmark-cpu.m	29
6.2	banchmark-gpu.m	30
6.3	banchmark-koz-cpu.m	31
8.1	SharpnessCore.java	41
8.2	BrightnessMatrix.java	42
8.3	FTHighKernelMatrix.java	43
8.4	ConvolutionMatrix.java	43
8.5	FTLowKernelMatrix.java	44
8.6	sumSharpness.java	44
8.7	sumOverSharpness.java	44
8.8	queue.sql	45
8.9	servlet.sql	45
9.1	ImageComparatorWithConstantScale.java	49

SEZNAM PŘÍLOH

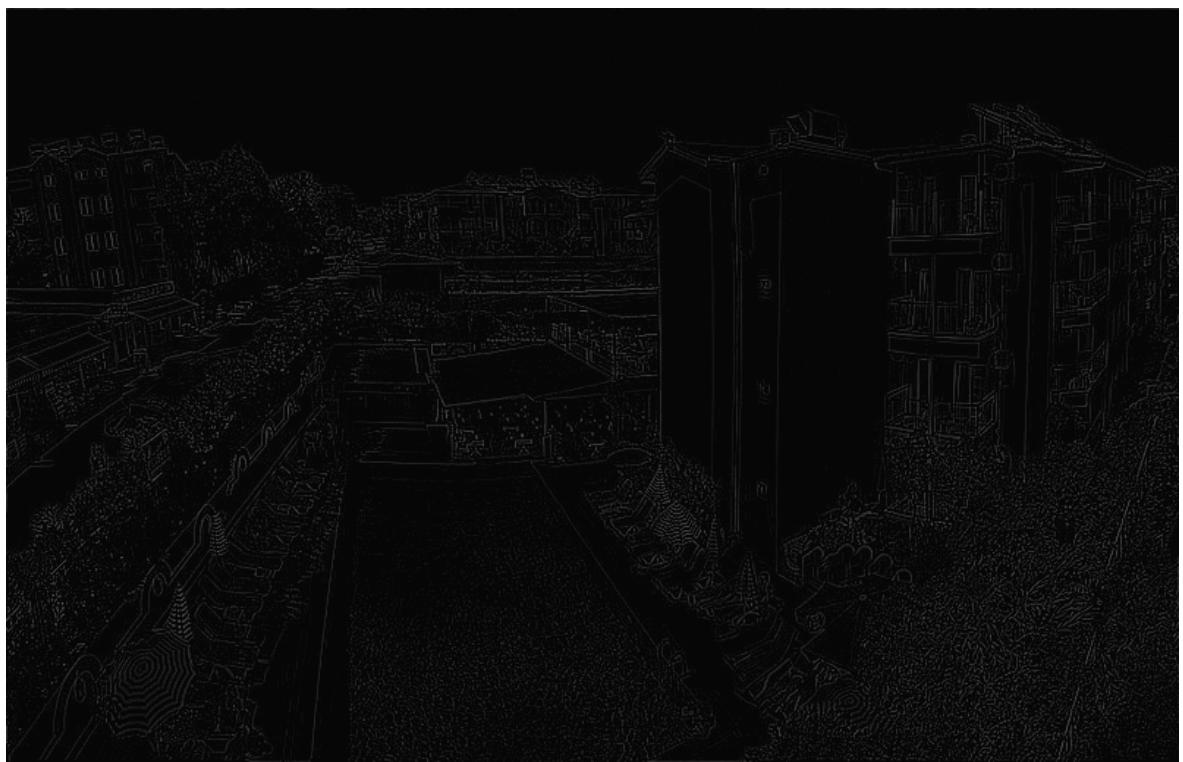
- P I. Ilustrační fotografie pro příklad přeostřenosti (originál)
- P II. Ilustrační fotografie pro příklad přeostřenosti (hrany)
- P III. Ilustrační fotografie pro příklad přeostřenosti (přeostřené hrany)

PŘÍLOHA P I. ILUSTRAČNÍ FOTOGRAFIE PRO PŘÍKLAD PŘEOSTŘENOSTI (ORIGINÁL)



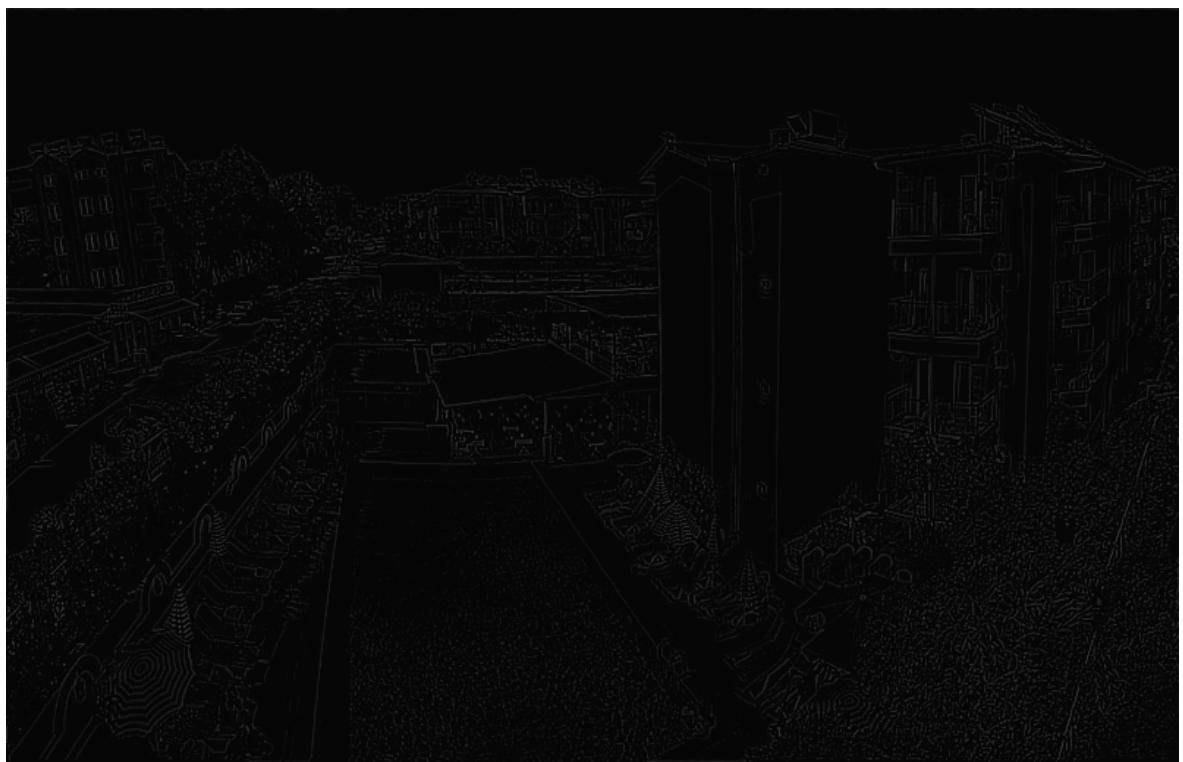
Obr. 1.1 Fotografie před úpravou

PŘÍLOHA P II. ILUSTRAČNÍ FOTOGRAFIE PRO PŘÍKLAD PŘEOSTŘENOSTI (HRANY)



Obr. 2.1 Vizualizace matice hran

PŘÍLOHA P III. ILUSTRAČNÍ FOTOGRAFIE PRO PŘÍKLAD PŘEOSTŘENOSTI (PŘEOSTŘENÉ HRANY)



Obr. 3.1 Vizualizace matice přeostřených hran